

Studies on Enumeration of
Acyclic Substructures
in Graphs and Hypergraphs
(グラフや超グラフに含まれる非巡回部分構造の列挙に関する研究)

Kunihiro Wasa

February 2016

Division of Computer Science

Graduate School of Information Science and Technology

Hokkaido University

Abstract

Recently, due to the improvement of the performance of computers, we can easily obtain a vast amount of data defined by graphs or hypergraphs. However, it is difficult to look over all the data by mortal powers. Moreover, it is impossible for us to extract useful knowledge or regularities hidden in the data. Thus, we address to overcome this situation by using computational techniques such as data mining and machine learning for the data. However, we still confront a matter that needs to be dealt with the exponentially many substructures in input data. That is, we have to consider the following problem: given an input graph or hypergraph and a constraint, output all substructures belonging to the input and satisfying the constraint without duplicates. In this thesis, we focus on this kind of problems and address to develop efficient enumeration algorithms for them.

Since 1950's, substructure enumeration problems are widely studied. In 1975, Read and Tarjan proposed enumeration algorithms that list spanning trees, paths, and cycles for evaluating the electrical networks and studying program flow. Moreover, due to the demands from application area, enumeration algorithms for cliques, subtrees, and subpaths are applied to data mining and machine learning. In addition, enumeration problems have been focused on due to not only the viewpoint of application but also of the theoretical interest. Compared to other algorithms, enumeration algorithms have some unique aspects. Suppose that the size of the input is n , the size of the outputs may be 2^n and thus the enumeration algorithm runs in $O(2^n)$ time. However, the size

is typically much smaller than the 2^n , and hence $O(2^n)$ is overestimation. Therefore, we consider the complexity of enumeration algorithms with respect to the size of *inputs and outputs*. Particularly, we say an enumeration algorithm runs polynomial *delay* if the algorithm outputs two consecutive solutions in polynomial time in the worst case.

Our goal of this thesis is to develop an efficient enumeration algorithm for acyclic substructures of graphs and hypergraphs with respect to the delay of the algorithm. Acyclic substructures are widely used in the field of Bioinformatics, data mining, and the design of the scheme of databases. In the following, we give the problems considered in this thesis. In Chap. 3, we consider the k -subtree enumeration problem originally introduced by Ferreira, Grossi, and Rizzi in 2011. This problem defined as follows: given an input graph G and an integer k , output all connected and acyclic subgraphs with k vertices in G , called k -subtree, without duplicates. They developed an enumeration algorithm that runs in $O(k)$ amortized time per solution, that is, their algorithm needs $O(k)$ time to output a solution after outputting the previous solution on average. However, whether this problem can be solved in constant amortized time per solution is still open. As a result of this thesis, we develop an optimal enumeration algorithm that runs in constant delay when we restrict inputs to trees by using the reverse search technique developed by Avis and Fukuda. We achieve this time complexity by define two types of the parent child relationships between two k -subtrees such that (I) a parent and its children have the same root, and (II) a parent and its children have the different root. This is the first optimal enumeration algorithm for the class of k -subtree problems. In Chap. 6, we consider the computation of the tree similarity between two given trees. As an application of our k -subtree enumeration algorithm, we propose an algorithm for the tree similarity that runs with the same time complexity of our algorithm.

Until now, problems for subgraph enumeration are widely studied. However, induced graph enumeration problems even for fundamental structures like trees are not

studied well. In Chap. 4, we consider an enumeration problem for connected and acyclic induced graphs, called *induced trees*, in a graph. It is difficult to overcome the barrier of $O(d)$ delay by naively applying the reverse search technique or binary partition method to develop an enumeration algorithm, where d is the maximum degree of an input graph. To develop an efficient algorithm, we focus on the degeneracy of graphs. The degeneracy of a graph G is k if every induced graph of G has a vertex whose degree is at most k . There exists some graph classes whose degeneracy are constant. For example, the degeneracy of trees, grid graphs, and planar graphs are at most 1, 2, and 5, respectively. In addition, if the degeneracy of G is k , then there exists a vertex ordering (v_1, \dots, v_n) of G that satisfies $\forall i \in [1, n], |v_j|_i < j \wedge v_j \in N(v_i)$. By using this ordering, we develop an enumeration algorithm for induced tree enumeration problem. The proposed algorithm recursively enumerates all solutions from the empty induced tree by adding vertices according to the ordering. This algorithm runs in $O(k)$ amortized time per solution for general graphs and is optimal when the degeneracy of an input graph is constant.

In Chap. 5, we consider an enumeration problem for acyclic subhypergraphs in a hypergraph. Hypergraphs that generalizes graphs are known as an important concept for designing a database scheme. Acyclicities for hypergraphs are defined hierarchically, called Berge-, α -, β -, and γ -acyclicity. Hitherto, an enumeration algorithm that enumerates maximal α -acyclic subhypergraph in an input hypergraph is proposed by Daigo and Hirata, however, other enumeration algorithms for acyclic subhypergraphs are hardly studied. In this chapter, we focus on Berge-acyclic subhypergraphs and develop an enumeration algorithm for them. Our proposed algorithm runs in $O(|S| + |N(S)|)$ time per solution S , where $N(S)$ the set of hyperedges incident to S and $|S| := \sum_{s \in S} |s|$. This algorithm is the first enumeration algorithm for Berge-acyclic subhypergraphs that runs in polynomial delay. In Chap. 7, we give the conclusion and future directions of this study.

Acknowledgement

I would like to express my special appreciation and thanks to my adviser Hiroki Arimura. He not only support my life in Hokkaido university but also advise me how to behave as a researcher. I can not finish writing this thesis without his kind support. I would like to express thanks to associate professor Takuya Kida. He always gave advice to me when I had a trouble. My co-authors, Ryohei Fujimaki, Kouichi Hirata, Kimihito Ito, Kouta Iwadate, Yusaku Kaneta, Yukitaka Kusumura, Takeaki Uno, and Katsuhisa Yamanaka always kindly supported during writing our papers. Ms. Manabe who is a secretary of our laboratory always helped us with our daily life in our laboratory. The member of our laboratory always make our laboratory good atmosphere. Last but not least, I would like to express my gratitude to my family. I could not imagine that I had still been a student more than 20 years. However, they have supported me such a long time.

Contents

1	Introduction	1
1.1	Background	1
1.2	Contributions	2
1.2.1	K -subtree enumeration	3
1.2.2	Induced tree enumeration	5
1.2.3	Berge-acyclic subhypergraph enumeration	6
1.3	Organization	6
2	Preliminaries	9
2.1	Graphs and Trees	9
2.1.1	Graphs	9
2.1.2	Trees	10
2.2	Hypergraphs	11
2.3	Enumeration Algorithms	11
2.3.1	Complexity of enumeration algorithms	12
2.3.2	Basic techniques	12
2.4	Related Work	15
2.4.1	Spanning tree enumeration	15
2.4.2	Subtree enumeration	16
2.4.3	Path enumeration	16

3	<i>K</i>-subtree Enumeration	21
3.1	Preliminaries	22
3.1.1	DFS-numbering	22
3.1.2	<i>K</i> -subtrees and their properties	23
3.1.3	<i>K</i> -subtree enumeration problem in a tree	25
3.2	The Parent-child Relationship Among <i>K</i> -subtrees	26
3.2.1	Basic idea: a family tree	27
3.2.2	Traversing <i>k</i> -subtrees	28
3.2.3	A intra-family tree for non-serial <i>k</i> -subtrees	29
3.2.4	A inter-family tree for serial <i>k</i> -subtrees	30
3.2.5	Putting them together	31
3.3	The Constant Delay Enumeration Algorithm	32
3.3.1	Generation of non-serial <i>k</i> -subtrees	32
3.3.2	Generation of serial <i>k</i> -subtrees	34
3.3.3	The proposed algorithm	35
3.4	Application to the Graph Motif Problem for Trees	39
4	Induced Tree Enumeration	45
4.1	Preliminaries	47
4.1.1	<i>K</i> -degenerate graphs	47
4.2	Basic Binary Partition Algorithm	48
4.2.1	Candidate sets and forbidden sets	48
4.2.2	Basic binary partition	50
4.3	Improved Binary Partition Algorithm	51
5	Berge-Acyclic Subhypergraph Enumeration	57
5.1	Preliminaries	59
5.1.1	Berge-acyclicity	59

5.1.2	Other Definitions and Properties	61
5.2	The Basic Algorithm	64
5.3	The Modified Algorithm	69
6	K-subtree Bit Enumeration	75
6.1	Preliminaries	76
6.1.1	Bit signatures	76
6.1.2	Tree similarity	77
6.2	Enumeration of At-Most K -subtrees in a Tree	78
6.2.1	The outline of the enumeration algorithm	79
6.2.2	Fast update of bit signatures	81
6.3	Enumeration of Exact K -subtrees in a Tree	83
6.3.1	The outline of the algorithm	83
6.3.2	Fast update of bit signatures	83
6.4	Experiments	88
6.4.1	Comparison of algorithms on real data	89
6.4.2	Comparison of algorithms on artificial data	89
6.4.3	Computing the gram matrix of a set of trees	89
6.4.4	Summary of experimental results	90
7	Conclusion and Future Work	93

List of Figures

3.1	Example of a family tree for k -subtrees	24
3.2	The bad case for Case 3 $\beta \in Ch(\ell)$ in the proof for Lemma 4	40
3.3	An idea of the proof for Lemma 5	40
4.1	Example of an induced tree	49
4.2	Example of a vertex ordering satisfying the condition (\star) of the degeneracy	49
4.3	Changing between candidate set $CAND$ after adding a vertex	55
5.1	Example of a hypergraph	63
5.2	Example of an incident matrix for a hypergraph	63
6.1	Example of a histogram of k -subtrees of a tree	80
6.2	The result of an experiment on the real phylogenetic tree	91
6.3	The result of an experiment on the artificial tree	91

Chapter 1

Introduction

1.1 Background

By emergence of massive structured data in the form of graphs and hypergraphs, there have been increasing demands on methods that discover many of interesting substructures called *patterns*, or regularities hidden in collections of such structured data. Especially, the data mining area has a large amount of studies on pattern mining problem such as frequent itemset mining [145, 162, 160, 161], sequence mining [2, 3, 111], and trees and graph mining [4, 68, 80, 157]. However, we are confronted with a huge amount of patterns including non-interesting ones when we try to obtain interesting patterns by using such mining techniques. Thus, there are great needs of the algorithm theory on enumeration problems.

Enumeration problems dealt with this thesis are the following problems: given a discrete structure and some constraints, output all substructures that satisfy the given constraints and belong to the given structure without duplicates. For example, suppose that a problem whose task is given an integer set $N = \{1, 2, 3, 4, 5, 6\}$ and $k = 4$, enumerate all subsets of S with k elements. This problem demands to output $\mathcal{S} = \{\{1, 2, 3, 4\}, \{1, 2, 3, 5\}, \dots, \{3, 4, 5, 6\}\}$. Because of both practical and theoretical

interests, enumeration problems have been widely studied since the last century. Until now, more than 500 references have been published¹, for example enumeration for spanning trees [156, 127, 142, 97, 40, 138, 124, 101, 38, 76, 56, 49, 72, 79, 57, 55, 113, 100, 102, 19, 20, 125], cliques [134, 105, 1, 17, 74, 59, 115, 27, 78, 110, 128, 95, 70, 34, 35, 82, 132, 18, 104, 109, 62, 114, 69, 58, 67, 123, 139, 41, 63, 25, 36, 26, 64, 22], matchings [24, 60, 54, 52, 98, 137, 136, 14, 8], independent sets [134, 103, 93, 90, 73, 78, 23, 10, 15, 39, 16, 108, 107], transversal on hypergraphs [14, 129, 32], necklaces on strings [50, 51, 119, 148, 117, 21, 118, 122], well-formed parenthesis [147, 42], and so on. However, there exist a lot of problems whose optimal enumeration algorithms are not obtained.

On the other hands, we also focus on acyclic substructures of structured data. Recently, we have obtained a huge amount of *acyclic* structures that are structures with no cyclic substructures, such as the second structures of RNAs and phylogenetic trees in biology, semi-structured text data XML and JSON in the web application, parsing trees in natural language processing, and so on. Compared to simple structured data like strings, such structures are complicated. For example, the edit distance between strings can be easily computed [135]. However, the computation of edit distance between rooted, labeled, and unordered trees is NP-hard [12]. In spite of their structural complexity, researchers struggle to reveal the character of acyclic structures since, for example, if a database is acyclic structure, some hard decision problems of the database can be solved in polynomial time [43].

1.2 Contributions

Our goal of this study is to develop an efficient enumeration algorithm for acyclic substructures of graphs and hypergraphs. Compared to other algorithms, enumeration

¹You can find a part of the references for enumeration problems in [149].

algorithms have some unique aspects. Suppose that the size of the input is n , the size of the outputs may be 2^n and thus the enumeration algorithm runs in $O(2^n)$ time. However, the size is typically much smaller than the 2^n , and hence $O(2^n)$ is overestimation. Therefore, we consider the complexity of enumeration algorithms with respect to the size of *inputs and outputs*. Particularly, we say an enumeration algorithm runs polynomial delay if the algorithm outputs two consecutive solutions in polynomial time. The precise definition of the complexity can be found in Sect. 2.3.1. In the following, we give the problems considered in this thesis and the contributions.

1.2.1 K -subtree enumeration

A k -subtree problem originally introduced by Ferreira, Grossi, and Rizzi [46] is the following problem: given a graph G and an integer k , output all k -subtrees in G without duplicates, where a k -subtree is a connected and acyclic edge-induced subgraph with k vertices of G . They developed an enumeration algorithm that runs in $O(k)$ amortized time per solution. The k -subtree enumeration problem considered in this thesis is closely related to an well-known graph problem of enumerating all spanning trees in an undirected graph G [113] since spanning trees are $k = n$ -subtrees. For this problem, Read and Tarjan [113] first presented an $O(ns + m + n)$ time and $O(m + n)$ space algorithm in 1970's, where s is the number of solutions, m is the number of edges in G , and n is the number of nodes in G . Recently, Shioura, Tamura, and Uno [125] presented $O(m + n + s)$ time and $O(m + n)$ space algorithm. Now the following question naturally arises as to whether there exists a constant delay enumeration algorithm for a k -subtree problem. A spanning tree enumeration algorithm presented by Shioura *et al.* is a such algorithm when $k = n$. However, it is not easy to extend the algorithms for spanning tree enumeration to k -subtree enumeration when $k \neq n$.

In Chap. 3, we propose an optimal algorithm for a k -subtree enumeration problem when we restrict its inputs to trees. The proposed algorithms is based on on the reverse

search technique [7]. One of the important key to develop enumeration algorithm using this technique is to define a nice parent-child relationship among the solutions. By defining two kinds of parent-child relationship among k -subtrees in a tree based on the numbering vertices in the tree according to the depth first manner, we achieve that our algorithm runs in constant worst-case time per solution with linear space and linear time preprocessing².

K -subtree bit enumeration

Similarity search is a fundamental problem in modern information and knowledge retrieval [96]. In particular, we focus on a *tree similarity* between two trees, which plays a key role in a number of information and knowledge retrieval problems from semi-structured data such as similarity search, clustering, recommendation, and classification for structured data in the real world [29, 28, 89, 6, 77]. A *frequency based* of tree similarity has been widely studied so far. Some substructures can be efficiently counted like paths [81] and q -grams [85]. On the other hands, Kashima and Koyanagi [77] presented an efficient dynamic programming algorithm to compute the *ordered subtree kernel* of two ordered trees using general ordered subtrees of *unbounded size*. However, it seems to be difficult to apply their algorithm to general ordered subtrees of *bounded size*.

In this thesis, we study efficient computation of tree similarity between two ordered trees using as features the class of *k -subtrees in unrestricted shape*. As a result, we show two efficient algorithms for computing the tree similarity by using k -subtree enumeration problem we developed³. We note that our new algorithms are the first *compressed pattern enumeration algorithms* [155] for a subclass of trees and graphs.

²This result has been published in [152].

³This result has been published in [150].

1.2.2 Induced tree enumeration

Compared to subgraph enumeration problems, induced subgraph enumeration problems are not studied well. Avis and Fukuda [7] showed a k -vertex connected induced subgraphs can be enumerated in $O(nm)$ time per solution using the reverse search technique. Recently, Ferreira [45] showed a k -vertex connected induced subgraphs can be enumerated in $O(\sum_{S \in \mathcal{S}} |E[S]|)$ total time, where \mathcal{S} is the set of solutions and $|E[S]|$ is the number of edges in S . Uno and Satoh[146] showed an enumeration algorithm for induced cycles that runs in $O(m)$ amortized time per solution. Ferreira *et al.* [47] improved Uno and Satho's algorithm. Their algorithm [47] runs in $\tilde{O}(n)$ amortized time per solution. However, a non-trivial enumeration algorithm for induced trees are not known.

In this thesis, we consider an induced tree enumeration problem. It is difficult to overcome the barrier of $O(d)$ delay by naively applying the reverse search technique or binary partition method to develop an enumeration algorithm, where d is the maximum degree of an input graph. To overcome this difficulty, we focus on the degeneracy of graphs. The degeneracy of a graph G is k if every induced graph of G has a vertex whose degree is at most k . There exists some graph classes whose degeneracy are constant. For example, the degeneracy of trees, grid graphs, and planar graphs are at most 1, 2, and 5, respectively. In addition, if the degeneracy of G is k , then there exists a vertex ordering (v_1, \dots, v_n) of G that satisfies $\forall i \in [1, n], |v_j|_i < j \wedge v_j \in N(v_i)$. By using this ordering, we develop an enumeration algorithm for induced tree enumeration problem that runs in $O(k)$ amortized delay⁴. Note that the algorithm is an optimal enumeration algorithm when the degeneracy of an input graph is constant such as planer graphs.

⁴This result has been published in [151].

1.2.3 Berge-acyclic subhypergraph enumeration

An *acyclic* sub-hypergraph is a generalization of the notion of subtrees for graphs, which means the hypergraph contains no cycle of connecting hyperedges. In the example of Fig. 5.1, the subset consisting of hyperedges 1, 2, 4, and 5 forms such a connected and acyclic sub-hypergraph. Particularly, we consider Berge-acyclicity [11], which locates the bottom of the degrees among other notions of acyclicities such as α -, β -, and γ -acyclicities. A Berge-acyclic graph has tree-like shape consisting of hyperedges. Thus, mining of such connected and Berge-acyclic sub-hypergraphs in a hypergraph can be regarded as finding less redundant subsets of groups reachable by chains of neighbor relations.

In this thesis, we consider an enumeration problem for Berge-acyclic sub-hypergraphs. As a result, we will develop an efficient enumeration algorithm for this problem by using the reverse search technique. This algorithm runs in $O(|S| + |N(S)|)$ time per solution S , where $|N(S)|$ is the size of S and its neighbor hyperedges. This is the first polynomial delay algorithm for this problem⁵.

1.3 Organization

In Chap. 2, we give the basic notations used in this thesis. In Chap. 3, we present the optimal enumeration algorithm for k -subtree enumeration problem. In Sect. 3.2, we first introduce the family tree for k -subtrees in a tree, and in Sect. 3.3, then, we present a constant delay algorithm that solves the k -subtree enumeration problem. Section 3.4 gives an application to the graph motif problem.

In Chap. 4, we give the induced tree enumeration algorithm. The algorithm is an optimal algorithm when the degeneracy of an input graph is constant. In Sect. 4.2, we propose a basic enumeration algorithm based on a binary partition method. In

⁵This result has been published in [153].

Sect. 4.3, we improve the algorithm by using a property of the degeneracy, and analyze its time complexity.

In Chap. 5, we give the Berge-acyclic sub-hypergraph enumeration algorithm. In Sect. 5.1, we give basic definitions and notations on hypergraphs and our data mining problem. In Sect. 5.2, we present the basic depth-first algorithm **BergeEnum** for the problem. In Sect. 5.3, we present the modified version of the algorithm, **FastBergeEnum**, using incremental computation.

As an application of the result of Chap. 3, in Chap. 6, we also present the k -subtree enumeration algorithm that enumerates bit signatures of k -subtrees. In Sect. 6.1.1, we introduce the bit signature of ordered k -subtrees, and in Sect. 6.1.2, we give the definitions of the tree similarities with ordered k -subtrees. In Sect. 6.2, we present the first algorithm using at most k -subtrees, and in Sect. 6.3, the second algorithm using exactly k -subtrees. In Sect. 6.4, we ran experiments to evaluate these algorithms.

In Chap. 7, we summarize this thesis. In this chapter, we also point out the future direction of this study.

Chapter 2

Preliminaries

In this chapter, we introduce the terminologies used in this thesis. For a set S , we denote by $|S|$ the number of elements in S . For mutually disjoint sets X and Y , we denote by $X \uplus Y$ the disjoint union of X and Y . For every integers $i \leq j$, we denote by $[i, j] = \{i, i + 1, \dots, j\}$. We write $S - e$ and $S + e$ for $S \setminus \{e\}$ and $S \cup \{e\}$, respectively. As a computation model, we adopt the usual RAM [30].

2.1 Graphs and Trees

2.1.1 Graphs

Let $G = (V(G), E(G))$ be an *undirected graph*, or simply a *graph*, where $V(G)$ is the set of *vertices* of G and $E(G) \subseteq V(G)^2$ is the set of *edges* of G . We denote by n and m the cardinality of $V(G)$ and $E(G)$, respectively. We denote by (u, v) in $E(G)$ the edge connecting vertices u and v in $V(G)$. We say that u and v are *adjacent* to each other if $(u, v) \in E(G)$. We denote by $N_G(u)$ the set of all vertices adjacent to u in G . We define the *degree* $d_G(u)$ of u in $V(G)$ as the number of vertices adjacent to u . The edge (u, v) is a *loop* if $u = v$. G has *parallel edges* if G has two distinct edges $e = (u, v)$ and $f = (s, t)$ such that $u = s \wedge v = t$ or $u = t \wedge v = s$. In this thesis, we assume that G is

simple, i.e., there is no loop and no parallel edges. We also assume that G is finite. In what follows, if it is clear from context, we omit the subscript G .

A *path* in G is a sequence of distinct vertices $\pi(u, v) = (v_1 = u, \dots, v_j = v)$, such that v_i and v_{i+1} are adjacent to each other for $1 \leq i < j$. If there is $\pi(u, v)$ in G , we say that the path *connects* u and v . G is *connected* if there is a path connecting any pair of vertices in G . The *length* of $\pi(u, v)$ is the number of edges in $\pi(u, v)$. We denote by $|\pi(u, v)|$ the length of $\pi(u, v)$. For any path $\pi(u, v)$ of length larger than two, $\pi(u, v)$ is called a *cycle* if $u = v$. G is *acyclic* if G has no cycle.

A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V(G)$ and $E' \subseteq E(G) \cap (V' \times V')$. Let S be a subset of $V(G)$. $G[S] = (S, E[S])$ denotes the graph *induced* by S , where $E[S] = \{(u, v) \in E(G) \mid u, v \in S\}$. We call $G[S]$ the *induced subgraph* of G . If no confusion, we identify S with $G[S]$ since we can uniquely determine $G[S]$ by S . A *connected component* is a maximal connected induced subgraph of G in terms of a vertex inclusion.

2.1.2 Trees

A *rooted tree* is a connected acyclic graph $T = (V(T), E(T), \text{root}(T))$, where $V(T)$ is the set of vertices, $E(T) \subseteq V^2$ is the set of edges, and $\text{root}(T) \in V(G)$ is a distinguished vertex, called the *root* of T . For each edge $(u, v) \in E(T)$, if $|\pi(u, \text{root}(T))| < |\pi(v, \text{root}(T))|$, we call u the *parent* of v and v a *child* of u . Since T is connected and acyclic, for any vertex $v \in V(T)$, there exists the unique path from v to the root. Thus, each vertex has the unique parent. For each vertex v , we denote the unique parent of v by $\text{pa}(v)$, and the set of all children of a vertex u by $\text{Ch}(u) = \{w \in V \mid (u, w) \in E\}$. A vertex v is a *leaf* if v has no child. We denote by $\text{Lv}(T) \subseteq V(T)$ the set of all leaves of T . We say that vertices u and v are *siblings* each other if they have the same parent. We call T is an *ordered tree* if we give a left-to-right order among siblings in T . We denote by \mathcal{T} the countable set of all ordered trees.

We define the ancestor-descendant relation \preceq as follows: For any pair of vertices u and $v \in V$, if there is a path $\pi = (v_0 = r, \dots, v_i = u, \dots, v_k = v)$ ($0 \leq i \leq k$), then we define $u \preceq v$, and say that u is an *ancestor* of v , or v is a *descendant* of u . If $u \preceq v$ but $u \neq v$, then this relationship is denoted by $u \prec v$, and u is a *proper ancestor* of v , or v is a *proper descendant* of u . For any vertex v , we denote by $T(v)$ the *set of all descendants* of v in T . We call $T(v)$ the *subtree* of T rooted at $v \in T$. The *border set* is the set $Bd(S) = \{y \in Ch(x) \mid x \in S, y \notin S\}$, that is, the set of all vertices that are not contained in S , but are children of some vertices in S .

2.2 Hypergraphs

A *hypergraph* is any pair $\mathcal{H} = (V, \mathcal{E}) = (V(\mathcal{H}), \mathcal{E}(\mathcal{H}))$ consists of the following two components: (I) the set of *vertices* $V = \{1, \dots, n\}$, $n \geq 0$, and (II) the set of *hyperedges* $\mathcal{E} = \{e_1, \dots, e_m\}$, $m \geq 0$, where for every $1 \leq i \leq m$, the hyperedge e_i is any subset e_i of V , and its index i is called the *edge ID* of e_i . A subset of hyperedges in \mathcal{H} is just a subset $\mathcal{S} \subseteq \mathcal{E}(\mathcal{H})$, where the underlying vertex set is denoted by $V(\mathcal{S}) = \bigcup \mathcal{S} := \{x \in V \mid x \in e, e \in \mathcal{S}\}$. The *total size* of \mathcal{S} is defined by the sum $||\mathcal{S}|| = \sum_{e \in \mathcal{S}} |e|$ of the sizes of its members. In this way, we refer to any subset $\mathcal{S} \subseteq \mathcal{E}(\mathcal{H})$ as a *sub-hypergraph* of \mathcal{H} meaning $(V(\mathcal{S}), \mathcal{S})$ if it is clear from context. The definition of a sub-hypergraph in this thesis is also referred to as a *partial hypergraph* in literature.

2.3 Enumeration Algorithms

In this section, we introduce how to evaluate enumeration algorithms. We also introduce basic ideas for developing enumeration algorithms.

2.3.1 Complexity of enumeration algorithms

As mentioned in Introduction, enumeration problems deal with a huge amount of solutions, thus these has unique aspects in terms of the time complexity. Let $N = ||I||$ and $M = |S(I)|$ be the input and the output size on I , respectively. In the following, $poly(\cdot)$ be an arbitrary polynomial function. Enumeration involves a huge number of solutions, thus an enumeration algorithm \mathcal{A} is supposed to run in short time, with respect to the number of solutions N . \mathcal{A} is *output-sensitive* if the total running time of \mathcal{A} is $O(f(M, N))$ time, where $f(\cdot)$ is an arbitrary computable function. If $f = poly$, we say that \mathcal{A} is an *output-polynomial algorithm* [130] or a *polynomial total time algorithm* [73]. \mathcal{A} is *incremental polynomial* [73] if given an input and the subset of solutions S , \mathcal{A} outputs another solution not in S in $O(poly(N, |S|))$ time if exists.

We say that \mathcal{A} is a *polynomial amortized delay algorithm* if the total running time of \mathcal{A} for computing all solutions on I is $O(poly(N)M)$ time. \mathcal{A} is *constant amortized delay algorithm* if $poly(N)$ is a constant. \mathcal{A} is of *polynomial delay* using preprocessing $poly(N)$ if the *delay*, which is the maximum computation time between two consecutive outputs, is bounded by $poly(N)$ after preprocessing in $poly(N)$ time. If the delay of \mathcal{A} is constant, \mathcal{A} is of *constant delay*. Note that if \mathcal{A} is of constant delay then \mathcal{A} is also of constant amortized delay. However, the converse does not hold. Since the recursion of enumeration algorithms is much more structured compared to optimization, we can develop a non-trivial amortized analysis. One of the advanced techniques is *Push Out amortization* developed by Uno [141]. Our goal here is to devise a non-trivial efficient algorithm in the complexities of polynomial delay and polynomial space.

2.3.2 Basic techniques

The common idea of our algorithms is as follows: firstly, construct an *enumeration tree* for the set of solutions, which is the tree-shaped search space covering the set,

then traverse it. During traverse the enumeration tree, our algorithms maintain data structures called the *border* for a current solution. The border for a current solution possesses elements that can be added or deleted from the current solution so that the algorithms can easily obtain the next solution.

In this thesis, we develop enumeration algorithms by using enumeration trees and the border technique. In what follows, we introduce three basic ideas for developing enumeration trees.

Binary partition method

A *binary partition algorithm* \mathcal{B} for an enumeration problem Π is a recursive enumeration algorithm. In each iteration, the algorithm divides the search space into two subspaces; the one including the solutions *has* an element e of an input I and the other including the solutions *does not have* e . This pseudo algorithm corresponds to the following set operation:

$$\mathcal{S}(S, X, I) = \mathcal{S}(S \cup \{e\}, X, I) \cup \mathcal{S}(S, X \cup \{e\}, I),$$

where $\mathcal{S}(S, X, I)$ is the set of solutions of I in which the every solution includes a set S and does not include a set X such that $S \cap X = \emptyset$, and $e \notin S \cup X$.

Next, we define the enumeration tree $\mathcal{ST} = (\mathcal{I}, \mathcal{E})$ for a binary partition method as follows:

- \mathcal{I} is the set of iterations during an execution of \mathcal{B} .
- For each iteration i and j , $(i, j) \in \mathcal{E}$ if j is called from i . We say i is the *parent* of j and j is a *child* of i .

We can easily see that each iteration has the unique parent. We say an iteration i is the *root* of \mathcal{ST} , if i has no parent. In the execution of \mathcal{B} , if we reach a leaf iteration

that has no child of \mathcal{ST} and the corresponding set S is a solution of Π , then we output S . This framework is relatively simple. Moreover, by using suitable pruning techniques and amortized analysis, we can develop an efficient enumeration algorithm.

Reverse search

The *reverse search technique* is a framework of enumeration algorithms developed by Avis and Fukuda [7]. Intuitively, an enumeration algorithm using the reverse search lists the solutions of an enumeration problem by traversing on the spanning forest on the underlying graph structure for the solutions.

We will give the abstract of the way of developing an enumeration algorithm using the reverse search. Assume that \mathcal{V} be the set of candidates of solutions and \mathcal{S} be the set of solutions. Note that $\mathcal{S} \subseteq \mathcal{V}$. In the reverse search technique, we first define (1) a *local search function* $f : \mathcal{V} \setminus \mathcal{S} \rightarrow \mathcal{V}$ and (2) an *adjacency-oracle* $Adj : \mathcal{V} \times N^+ \rightarrow \mathcal{V}$, where $N^+ = N \setminus \{0\}$. Next, we build a tree-shaped search space called the *family tree* $\mathcal{F} = (\mathcal{V}, \mathcal{PC}(\mathcal{V}))$ for an enumeration problem. We call $\mathcal{PC}(\mathcal{V})$ the *parent-child relationship*. For any $v, v' \in \mathcal{V}$, $(v, v') \in \mathcal{PC}(\mathcal{V})$ if $v = f(v')$ and there exists an integer k such that $Adj(v, k) = v'$. Next, by traversing on \mathcal{F} , we enumerate all solutions without duplicates. Avis and Fukuda also showed that if $f(\cdot)$ and $Adj(\cdot)$ can be computed in polynomial time, then the above enumeration algorithm runs in polynomial delay with polynomial space. The most important thing to use the reverse search is how to develop a *nice* family tree.

Gray code method

Gray code method is one of the fundamental framework for enumeration algorithms. Assume that \mathcal{S} be the set of solutions, and $\log |\mathcal{S}| < n$. First, we define a bijection function $gc : \mathcal{S} \rightarrow \{0, 1\}^n$, where $\{0, 1\}^n$ be the set of binary strings with length n . Then, an enumeration algorithm based Gray code traverses on the n -dimensional

hypercube. Each vertex of the hypercube corresponds to a binary string. During the traverse, the algorithm outputs all elements in the set of B such that for each $b \in \{0, 1\}^n$, $b \in B$ if and only if $(\exists s \in \mathcal{S})gc(s) = b$. Please consult the survey of Gray code method [121] for the detail.

2.4 Related Work

In this section, we give some remarkable results for enumeration problems for acyclic substructures, especially spanning trees, subtrees, and paths. There exist surveys of enumeration algorithms by Fukuda [53] and Yoshida [159]. We also enumerate enumeration algorithm on our web page [149].

2.4.1 Spanning tree enumeration

Spanning trees are the fundamental acyclic structures and their enumeration problems are widely studied. In 1961, Hakimi [61] developed an enumeration algorithm for spanning trees in an input graph. In 1965, Minty gave an enumeration algorithm for spanning trees of a given graph [102]. One decade later, Read and Tarjan [113] presented an $O(ns + m + n)$ time and $O(m + n)$ space algorithm, where s is the number of solutions. Kapoor and Ramesh gave a first constant amortized delay enumeration algorithm for spanning trees with polynomial space [76]. Shioura *et al.* showed a spanning tree enumeration problem can be solved in constant amortized time per solution with linear space [125]. There also exist enumeration algorithms for spanning trees [100, 57, 72, 124, 138, 97, 142], weighted spanning trees in non-decreasing order [55, 79, 49, 38, 127], and minimum weighted spanning trees [156].

There exists k minimum spanning tree enumeration algorithms. Gabow showed a k minimum spanning trees can be listed in $O(km\alpha(n, m) + m \log m)$ total time with $O(k + m)$ space [55]. Katoh *et al.* developed k minimum spanning tree enumeration

algorithm. Their algorithm runs in $O(km + \min(n^2, m \log \log n))$ total time and $O(k + m)$ space [79]. We summarize these results in Tab. 2.1.

2.4.2 Subtree enumeration

Subtrees are ones of the fundamental acyclic substructures. Ruskey gave an enumeration algorithm for subtrees in an input tree [116]. His algorithm runs in $O(n)$ time per solution, where n is the number of vertices in an input tree. Hikita gave an enumeration algorithm for non-isomorphic k -subtrees in an input tree [65], where a k -subtree is a subtree with k vertices. Recently, Ferreira *et al.* proposed a k -subtree enumeration algorithm for a graph [46]. Their algorithm runs in $O(k)$ amortized time per solution. In Chap. 3, we will show that, as a special case of Ferreira *et al.*'s problem, a k -subtree enumeration problem for a tree can be solved in constant delay. Wild showed that k -subtrees in a trees can be enumerated in $O(sn^5)$ total time by using a general enumeration algorithm for k -element belonging to a class of closure systems [154]. In Chap. 4, we will show that an induced tree enumeration problem for a graph can be solved in $O(k)$ amortize time per solution, where k is the degeneracy of an input graph. We summarize these results in Tab. 2.2.

2.4.3 Path enumeration

Ponstein showed that all paths in a graph can be listed by matrix computation [112]. Kamae gave an enumeration algorithm for directed paths [75]. Read and Tarjan gave the first output-polynomial time enumeration algorithm [113]. Birmelé *et al.* [13] showed st-path enumeration algorithm for a graph G that runs in $O(\sum_{\pi \in P_{st}(G)} |\pi|)$ total time where $P_{st}(G)$ is the set of st-paths in G . The time complexity of their algorithm does not exceed the time to list all st-paths in G asymptotically. Ferreira *et al.* [47] developed an enumeration algorithm for induced st-paths in a graph. Their algorithm is an improved version of the algorithm developed by Uno and Satoh[141].

Yau developed an enumeration algorithm for Hamiltonian paths [158]. Eppstein [37] gave an excellent survey on problems for finding k shortest paths. We summarize the part of results for path enumeration in Tab. 2.3.

Table 2.1: Spanning tree enumeration algorithms. Assume that n is the number of vertices the given graph, m is the number of edges, and s is the number of spanning trees.

Reference	time	space	method	note
All spanning trees in undirected graphs				
Hakimi [61]	-	-	Binary Partition.	
Minty [102]	$O(ms + m + n)$ total time	-	Binary Partition.	Analyzed by [113].
Mayeda and Seshu [100]	$O(nms + m + n)$ total time	$O(mn)$	Backtrack.	Analyzed by [113].
Read and Tarjan [113]	$O(ms + m + n)$ total time	$O(m + n)$	Backtrack.	
Gabow and Myers [57]	$O(ns + m + n)$ total time	$O(m + n)$	Binary Partition.	
Jayakumar <i>et al.</i> [72]	$O(n + m + n(s + t_0))$ total time	-		t_0 : number of non spanning tree
Kapoor and Ramesh [76]	$O(s + m + n)$ total time	$O(mn)$	Branch and bound.	
Shioura <i>et al.</i> [124]	$O(s + m + n)$ total time	$O(mn)$	Reverse search.	
Shioura <i>et al.</i> [125]	$O(s + m + n)$ total time	$O(m + n)$	Reverse search.	
Matsui [97]	$O(sn + m + n)$ total time	$O(m + n)$	Reverse search.	
All spanning trees in directed graphs				
Gabow and Myers [57]	$O(ms + m + n)$ total time	$O(m + n)$	Binary Partition.	
Kapoor and Ramesh [76]	$O(sn + n^3)$ total time	$O(n^2)$	Branch and bound.	
Uno [138]	$O(sD(n, m) + m)$ total time	$O(DS(n, m))$	Reverse search.	$D(n, m)$ and $DS(n, m)$ are the time and space complexities for the data structure.
Uno [142]	$O(s \log^2 n + m \log n + n)$ total time	$O(m + n)$	Binary Partition.	
K minimum spanning trees in non-decreasing order				
Burns and Haff [19]	-	-		
Camerini <i>et al.</i> [20]	-	-		
Gabow [55]	$O(km\alpha(n, m) + m \log m)$ total time	$O(k + m)$	Binary Partition.	
Katoh <i>et al.</i> [79]	$O(km + \min(n^2, m \log \log n))$ total time	$O(k + m)$	Binary Partition.	
Frederickson [49]	$O(k^2 \sqrt{m} + m \log \log_{(2+m/n)} n)$ total time	$O(m)$	Branch and bound.	$k = O(\sqrt{m})$.
Kapoor and Ramesh [76]	$O(k \log n + mn)$ total time	$O(n^2)$	Branch and bound.	
Eppstein [38]	$O(m \log \beta(m, n) + k^2)$ total time	-	Binary Partition.	
Sørensen and Janssens [127]	$O(km \log m + k^2)$ total time	$O(sm)$	Branch and bound.	
Minimum spanning trees				
Yamada [156]	$O(s(mn + n^2 \log n))$ total time	$O(m)$	Binary Partition.	

Table 2.2: Subtree enumeration algorithms. Assume that n is the number of vertices the given graph, m is the number of edges, and s is the number of solutions.

Reference	time	space	method	note
Subtrees with unbounded size				
Ruskey [116]	$O(s)$ total time	$O(m)$	Gray code.	
Subtrees with k vertices				
Hikita [65]	-	-	Gray code.	Non-isomorphic k -subtrees
Ferreira <i>et al.</i> [46]	$O(sk)$ total time	$O(m)$	Binary partition.	
Wild [154]	$O(sn^5)$ total time		Reverse search.	Inputs are trees.
Chap 3	$O(1)$ delay	$O(n)$	Reverse search.	Inputs are trees.
Induced trees with unbounded size				
Chap 4	$O(sk)$ total time	$O(n + m)$	Binary Partition.	k is the degeneracy of the input.

Table 2.3: Path enumeration. Assume that n is the number of vertices the given graph, m is the number of edges, and s is the number of solutions.

Reference	time	space	method	note
Path enumeration				
Ponstein [112]	-	-	Matrix computation.	
Danielson [33]	-	-	Matrix computation.	
Read and Tarjan [113]	$O(sm + n + m)$ total time	$O(n + m)$	Backtrack.	
Directed path enumeration				
Kamae [75]	-	-	Matrix computation.	
St-path enumeration				
Kroft [84]	-	-	Backtrack.	
Birmelé [13]	$O(\sum_{\pi \in P_{st}(G)} \pi)$ total time	$O(n + m)$	Binary partition.	$P_{st}(G)$ is the set of st-paths in G .
Induced path enumeration				
Uno and Satoh [141]	$O(n + m)$ time per solution	$O(n + m)$	Binary partition.	
Ferreira <i>et al.</i> [47]	$\tilde{O}(V)$ amortized time	$O(n + m)$	Binary partition.	
Hamiltonian path enumeration				
Yau [158]	-	-	Matrix computation.	

Chapter 3

K -subtree Enumeration

In this chapter, we consider a k -subtree enumeration problem, which is originally introduced by Ferreira *et al.* [46], where an instance consists of an undirected graph G of n vertices and a positive integer $k \geq 1$, and the task is to find all k -subtrees, a connected and acyclic subgraph consisting of exactly k vertices in G . Ferreira *et al.* [46] presented the first output-sensitive algorithm that lists all k -subtrees in a graph G in $O(sk)$ total time and $O(m)$ space, in other words, in $O(k)$ amortized time per subtree, where n is the number of vertices in G , m is the number of edges in G , and s is the number of solutions. As a special case of a k -subtree enumeration problem, a spanning tree enumeration problem ($k = n$) are widely studied. Shioura *et al.* showed a spanning tree enumeration problem can be solved in constant amortized time per solution with linear space [125]. However, it has been an open question whether there exists a faster enumeration algorithm that solves a k -subtree enumeration problem.

As a result, we present the first *constant delay enumeration algorithm* for the k -subtree enumeration problem in *trees*. More precisely, our algorithm lists all k -subtrees of an input tree T of size n in constant worst-case time per subtree using $O(n)$ preprocessing and space. Our algorithm is based on the reverse search technique, proposed by Avis and Fukuda [7]. However, unlike Ferreira *et al.*'s algorithm [46], our algorithm

achieves the best possible enumeration complexity.

One of our motivation comes from application to the graph motif problem (GMP, for short). Given a bag of k labels, called a *pattern*, and an input graph G , called a *text*, GMP asks to find a k vertex subgraph of G whose multi-set of labels is identical to a given pattern. Lacroix *et al.* [88] introduced the problem with application to biology and presented an FPT algorithm with $k = O(1)$, and NP-hardness in general. Then, Fellows *et al.* [44] showed that the problem is NP-hard even for trees of degree 3, and presented an improved FPT algorithm. Sadakane *et al.* [120] studied the string version of GMP, and presented linear-time algorithms. Although there are increasing number of studies on GMP [44, 88], there are few attempts to apply efficient enumeration algorithms to this problem. Ferreira *et al.* [46] mentioned above is one of such studies. Recent studies [6, 145, 161] in data mining applied efficient enumeration algorithms to discovery of interesting substructures from massive structured data in the real world.

3.1 Preliminaries

3.1.1 DFS-numbering

In the followings, we regard an input rooted tree T of size $n \geq 0$ as an ordered rooted tree. We number all vertices of T from 1 to n by the *DFS-numbering*, which is the preorder numbering in the depth-first search [30] on vertices in T . In what follows, we identify the vertex and the associated vertex number, and thus, write $V = \{1, \dots, n\}$. We write $u \leq v$ (resp. or $u < v$) if the numbering of u is smaller or equal to (resp. smaller than) that of v . As a basic property of a DFS-numbering, we have the next lemma.

Lemma 1. *For any $u, v \in V$, the DFS-numbering on T satisfies the following properties (i), (ii), and (iii):*

(i) If v is a proper descendant of u , i.e., $u \prec v$, then $u < v$ holds.

(ii) If v is a properly younger sibling of u , then $u < v$ holds.

(iii) Suppose that $u \not\leq v$ and $v \not\leq u$. For any vertices u' and v' such that $u \preceq u'$ and $v \preceq v'$, $u < v$ implies that $u' < v'$.

Proof. Properties (i) and (ii) are clear from the order of visiting vertices. For property (iii), since $u < v$, $u \not\leq v$, and $v \not\leq u$ hold, any vertices in $T(v)$ are visited after all vertices in $T(u)$ are visited. Furthermore, we see $u \leq u'$ and $v \leq v'$ because $u \preceq u'$ and $v \preceq v'$. Hence, $u' < v'$ holds. \square

3.1.2 K -subtrees and their properties

Let $1 \leq k \leq n = |T|$. A k -subset of T is any subset of $V(T)$ with k vertices. A subtree with size k of T , or simply a k -subtree of T , is any connected induced subgraph $T[S] = (S, E[S])$ of T induced in a k -subset $S \subseteq V(T)$ consisting of exactly k vertices of T . Note that since T is a tree, any connected subgraph forms a tree. We denote by $\mathcal{S}_k(T)$ the family of all k -subtrees of T . That is, $\mathcal{S}_k(T)$ is the family of k -subsets of T such that $S \in \mathcal{S}_k(T)$ forms a k -subtree of T . A k -subtree S appears in T if there is some subset $U \subseteq V(T)$ such that $T[U]$ is isomorphic to $T[S]$.

For a k -subtree S in T , we denote by $\text{root}(S)$ and $Lv(S)$ the root and the set of leaves of S , respectively. For subset S and its complement $\bar{S} = V(T) \setminus S$, we call an edge $e = (x, y)$ of T a cut edge between S and \bar{S} if $x \in S$ and $y \in \bar{S}$. The border set, denoted by $Bd(S)$, is the set of all lower ends y of cut edges (x, y) between S and \bar{S} defined by $Bd(S) = \{y \in V(T) \mid (x, y) \in E(T), x \in S, y \notin S\} = \{y \in Ch(x) \mid x \in S, y \notin S\}$. In other words, $Bd(S)$ is the set of all vertices lying immediately outside of S . We define the weight of a k -subtree S by the sum $w(S) = \sum_{v \in V(S)} v \geq 0$ of the DFS numbers of the vertices in S .

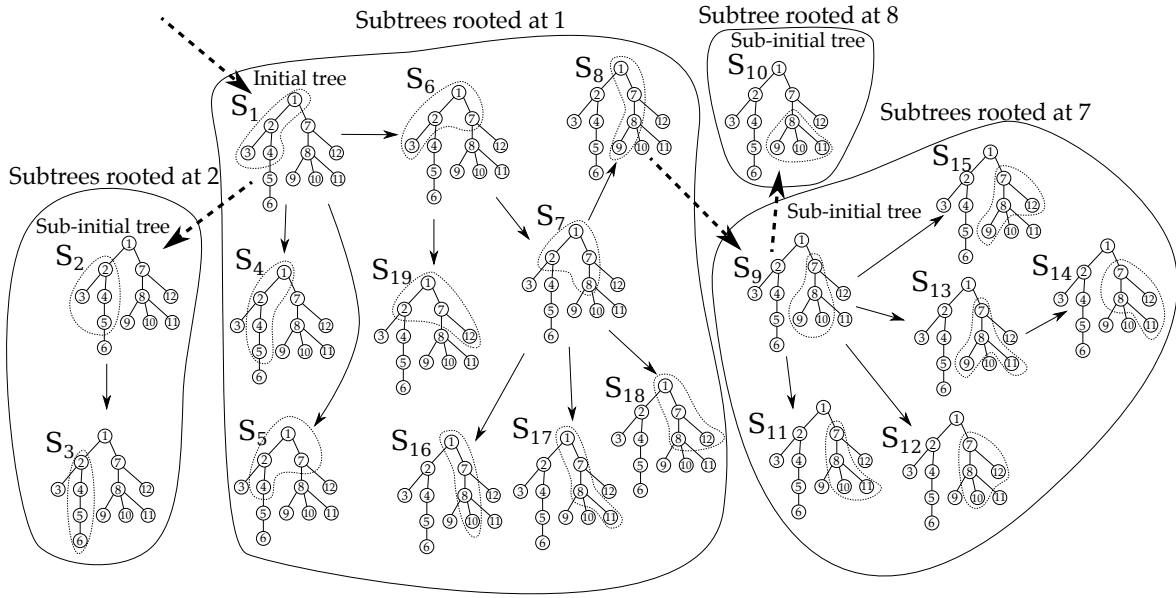


Figure 3.1: A family tree for all of nineteen k -subtrees of an input rooted tree T_1 of size $n = 11$, where $k = 4$. In this figure, each set of vertices surrounded by a dotted circle indicates a k -subtree, and each arrow (resp. dashed arrow) indicates the parent-child relation defined by the parent function \mathcal{P}^r for $r \in V(T_1)$ of type I (resp. \mathcal{P}^* of type II). We observe that the arrows among each set of subtrees in a large circle indicates a intra-family tree, and the dotted arrows among the set of the serial and pre-serial k -subtrees form the unique inter-family tree.

Next, we introduce a family of subtrees in special form, called serial trees, as follows. For any vertex r ($1 \leq r \leq n - k + 1$), the *serial k -subtree rooted at r* is the k -subtree $\mathcal{I}_k^r(T) = \{r, r + 1, \dots, r + k - 1\}$ in the DFS-numbering. A k -subtree S is *serial* if $S = \mathcal{I}_k^r(T)$ for some r , and S is *non-serial* otherwise.

Lemma 2 (DFS-numbering lemma). *For any k -subtree S in T , then*

- (i) *If S is non-serial, then $\min(\text{Bd}(S)) < \max(\text{Lv}(S))$ holds, and there exists a vertex v that satisfies $v \in B(S)$ and $\min(S) < v < \max(S)$.*
- (ii) *If S is serial and $\text{Bd}(S) \neq \emptyset$, then $\max(\text{Lv}(S)) < \min(\text{Bd}(S))$ holds.*

Proof. (i) If S is non-serial, then there is some $v \in V(T) \setminus S$ such that $\min(S) < v < \max(S)$. We can find some $v' \in B(S)$ such that $\min(S) < v' < \max(S)$ and $v' \preceq v$. Furthermore, if we take the smallest such v , then $v' = \min(\text{Bd}(S))$. Since $\max(\text{Lv}(S)) = \max(S)$, $\min(\text{Bd}(S)) < \max(\text{Lv}(S))$ holds. (ii) If S is serial, there is no border vertex between $\min(S)$ and $\max(S)$. Since any border vertex is a member of $T(\text{root}(S))$, it is properly larger than $\max(S)$. \square

3.1.3 K -subtree enumeration problem in a tree

Now, we state our problem below.

Problem 1 (k -subtree enumeration in a tree). *Given an input rooted tree T and a non-negative integer k , enumerate all the k -subsets of T without duplicates such that each of them forms k -subtrees of T .*

That is, it is the problem of enumerating all elements of $\mathcal{S}_k(T)$. Recall that $\mathcal{S}_k(T)$ is the set of k -subtrees appearing in T . The number of solutions is given as follows.

Lemma 3. *Let $s = f(k, n)$ be the number of all k -subtrees in an input rooted tree T of n vertices.*

- $s = O(n^k)$ for the upper bound.
- $s = 2^{\Omega(k)}$ for the lower bound.

Proof. For the upper bound, we can specify any k -subtree by selecting mutually distinct k vertices from n vertices of T . Thus, $s = f(k, n)$ is bounded from above by $\binom{n}{k} = O(n^k)$ for constant k . For the lower bound, we consider the infinite sequence of input rooted trees $\{T_k\}_{k \geq 1}$, where T_k is the rooted tree of height 1 and size $n = 2k - 1$ consisting of the root and $n - 1$ leaves only. Then, $s = f(k, n)$ is given by $\binom{n-1}{k-1}$. From the lower bound of binomial coefficient, this number is bounded from below

$$\binom{n-1}{k-1} \geq \left(\frac{n-1}{k-1}\right)^{k-1} = \left(\frac{2(k-1)}{k-1}\right)^{k-1} = 2^{k-1}.$$

Hence, we have $s = 2^{\Omega(k)}$. □

Our problem is a special case of the k -subtree problem in a graph, originally introduced and studied by Ferreira, Grossi, and Rizzi [46]. An input graph is a tree in our problem, while it is a general undirected graph in [46]. Ferreira *et al.* [46] showed an efficient enumeration algorithm that lists all k -subtrees in $O(k)$ amortized time per subtree for a general class of undirected graphs. However, its time complexity is still open when an input is restricted to rooted trees. Therefore, our goal is to devise an optimal algorithm that lists all k -subtrees in $O(1)$ worst-case time per subtree.

3.2 The Parent-child Relationship Among K -subtrees

Let us fix an input rooted tree $T = (V(T), E(T), \text{root}(T))$ with n vertices. We assume that vertices of T are numbered by the DFS-ordering. That is, $\text{root}(T) = 1$. Let $1 \leq k \leq n$ be any positive integer.

3.2.1 Basic idea: a family tree

Our algorithm is designed based on *the reverse search* technique by Avis and Fukuda [7]. In the reverse search technique, we define a tree-shaped search route on solutions, called a *family tree*.

A family tree for the family $\mathcal{S}_k(T)$ is a spanning tree $\mathcal{F}_k(T) = (\mathcal{S}_k(T), \mathcal{P}_k(T), \mathcal{I}_k(T))$ over $\mathcal{S}_k(T)$ as vertex set. In what follows, we write \mathcal{S} for $\mathcal{S}_k(T)$ by omitting T and k if no confusion. Similarly, we write \mathcal{F} , \mathcal{P} , \mathcal{I} , and so on. Given a family tree \mathcal{F} , we can enumerate all solutions traversing on \mathcal{F} from the root \mathcal{I} . The collection of *reverse edges* is given by a function $\mathcal{P} : \mathcal{S} \setminus \{\mathcal{I}\} \rightarrow \mathcal{S}$, called the *parent function*, that assigns the unique parent $\mathcal{P}(S)$ to each child k -subtree S except \mathcal{I} . Precise definitions of \mathcal{I} and \mathcal{P} will be given later. Note that a family tree may form a forest.

Example 1. In Fig. 3.1, we show an example of a family tree for all of nineteen k -subtrees of an input rooted tree T_1 of size $n = 11$, where $k = 4$.

A basic idea of the construction of the family tree \mathcal{F} is explained as follows. Recall that $V(T) = \{1, \dots, n\}$. First, we partition the family \mathcal{S} into the mutually disjoint sub-families $\mathcal{S} = \mathcal{S}_k^1(T) \uplus \dots \uplus \mathcal{S}_k^n(T)$, where for every $r \in V(T)$, the *sub-family* $\mathcal{S}^r = \mathcal{S}_k^r(T)$ is the set of all k -subtrees in T rooted at r .

The first task is to define the family tree $\mathcal{F}^r = \mathcal{F}_k^r(T)$, called an *intra-family tree*, for the traversal of all k -subtrees rooted at r that belongs to the sub-family \mathcal{S}^r . The root of the intra-family tree \mathcal{F}^r is the unique serial tree $\mathcal{I}^r = \mathcal{I}_k^r(T)$ in \mathcal{S}^r that consists of the vertex set $\{r, \dots, r + k - 1\}$. For the construction of \mathcal{F}^r , we define the parent function $\mathcal{P}^r = \mathcal{P}_k^r(T) : \mathcal{S}^r \setminus \{\mathcal{I}^r \mid r \in V(T)\} \rightarrow \mathcal{S}^r$ that uniquely assigns the parent $\mathcal{P}^r(S)$ with properly smaller weight to each non-serial k -subtree S rooted at r , called a k -subtree of type I. The construction of \mathcal{P}^r will be described in Sect. 3.2.3.

The second task is to define the family tree $\mathcal{F}^* = \mathcal{F}_k^*(T)$, called an *inter-family tree*, for the traversal between \mathcal{S}^r 's. The root of the inter-family tree \mathcal{F}^* is the unique

serial tree $\mathcal{I}^1 = \mathcal{I}_k^1(T)$ in \mathcal{S} that consists of the vertex set $\{1, \dots, k\}$. We define the parent function $\mathcal{P}^* = \mathcal{P}_k^*(T) : \{\mathcal{I}^r \mid r \in V(T)\} \setminus \mathcal{I}^1 \rightarrow \mathcal{S}$ that uniquely assigns the parent $\mathcal{P}^*(S)$ with properly smaller weight to each \mathcal{I}^r , that is, the serial k -subtree S rooted at r . The construction of \mathcal{P}^* will be described in Sect. 3.2.4.

Finally, we have the family tree $\mathcal{F} = (\mathcal{S}, \mathcal{P}, \mathcal{I})$ for the whole family \mathcal{S} by merging all intra-family trees and the inter-family tree, where the parent function \mathcal{P} is the disjoint union $\mathcal{P}^* \uplus \biguplus_r \mathcal{P}^r$, and the initial tree \mathcal{I} is the unique serial tree $\mathcal{I}_k^1(T)$ with the smallest weight. In the following sections, we will describe the details of the above construction.

3.2.2 Traversing k -subtrees

We efficiently traverse between two k -subtrees R and S in \mathcal{S} . Suppose we are to visit S from R . Then, we first delete a leaf $\ell \in Lv(R)$ from R , and next, add a border vertex $\beta \in Bd(R)$ to R . Unfortunately, this construction is not always sound, meaning that, sometimes, a certain combination of ℓ and β violates the connectivity condition on S . The next technical lemma precisely describes when this degenerate case happens and how to avoid it.

Lemma 4 (connectivity). *Let R be any k -subtree of T with size $k \geq 2$. Suppose that $\ell \in R$ and $\beta \notin R$ are any vertices of $T(\text{root}(R))$. Then, (i) and (ii) are equivalent:*

(i) *The set $S = (R \setminus \{\ell\}) \cup \{\beta\}$ is k -subtree.*

(ii) *$\ell \in Lv(R)$, $\beta \in Bd(R)$, and $\beta \notin Ch(\ell)$.*

Proof. (i) \Rightarrow (ii): By contradiction, we suppose that condition (ii) does not hold. Then, there are three cases below. (Case 1) $\ell \notin Lv(R)$: There dose not exist any vertex except ℓ such that the vertex is the parent of children of ℓ in T . Thus, S is not connected. (Case 2) $\beta \notin Bd(R)$: S consists of two or more connected sets such that one is $\{\beta\}$

and another is a set including the root vertex. (Case 3) $\beta \in Ch(\ell)$: See Fig. 3.2 for example. There is a parent-child relationship between β and ℓ in S , R , and T . Thus, S is obviously unconnected and is not a k -subtree. (ii) \Rightarrow (i): $S' = R \setminus \{\ell\}$ is obviously connected. Furthermore, $S = S' \cup \{\beta\}$ is connected since $\text{pa}(\beta) \in S'$. Thus, S is a k -subtree. \square

The next technical lemma is useful in showing the identity of two k -subtrees.

Lemma 5 (identity). *Let R be any k -subset of $V(T)$. Suppose that we take two k -subsets S and R' such that $S = (R \setminus \{\ell\}) \cup \{\beta\}$ and $R' = (S \setminus \{\beta'\}) \cup \{\ell'\}$ where $\ell \in R, \beta \notin R, \ell' \notin S$, and $\beta' \in S$. Then, we have the equivalence that (i) $R = R'$ holds if and only if (ii) $\ell = \ell'$ and $\beta = \beta'$ hold.*

Proof. First, (ii) \Rightarrow (i) is obvious (See Fig. 3.3). Therefore, we consider (i) \Rightarrow (ii). Suppose that (i) $R = R'$ holds. We assume that (ii) does not hold. There are two cases below. (Case 1) $\ell \neq \ell'$: In this case, R contains ℓ but not ℓ' , while R' contains ℓ' but not ℓ . Thus, R and R' can not be identical. (Case 2) $\beta \neq \beta'$: By symmetrically, R and R' can not be identical, too. By contradiction, (ii) holds. \square

3.2.3 A intra-family tree for non-serial k -subtrees

Firstly, for each vertex r in T , we describe how to build the intra-family tree \mathcal{F}^r for the subspace \mathcal{S}^r of all r -rooted k -subtrees of type I. Suppose that $|T(r)| \geq k$. Then, the intra-family tree $\mathcal{F}^r = (\mathcal{S}^r, \mathcal{P}^r, \mathcal{I}^r)$ is given as follows. The vertex set is the collection \mathcal{S}^r . The *sub-initial* k -subtree \mathcal{I}^r is given as a serial tree containing r as its root. Actually, such a serial k -subtree is uniquely determined by the k -subtree \mathcal{I}^r consisting of k vertices $\{r + i \mid i = 0, \dots, k-1\}$. Next, we give the parent function \mathcal{P}^r from $\mathcal{S}^r \setminus \{\mathcal{I}^r \mid r \in V(T)\}$ to \mathcal{S}^r as follows.

Definition 1 (the parent of k -subtree of type I). *Let T be an input rooted tree and $S \in \mathcal{S}^r \setminus \{\mathcal{I}^r \mid r \in V(T)\}$ be any non-serial k -subtree rooted at $r \in V(T)$. Then, the*

parent of S is the k -subtree

$$\mathcal{P}^r(S) = (S \setminus \{\ell\}) \cup \{\beta\}$$

obtained from S by deleting a vertex $\ell \in Lv(S)$ and adding a vertex $\beta \in Bd(S)$ satisfying the conditions that $\ell = \max(Lv(S))$ and $\beta = \min(Bd(S))$. Then, we say that S is a type-I child of $\mathcal{P}^r(S)$.

Lemma 6. *If $S \in \mathcal{S}^r \setminus \{\mathcal{I}^r \mid r \in V(T)\}$, then $\mathcal{P}^r(S)$ is uniquely determined, and an well-defined k -subtree of T . Furthermore, $w(\mathcal{P}^r(S)) < w(S)$ holds.*

Proof. Since S is non-serial, $\beta < \ell$ from Lemma 2. Then, we have $\beta \notin Ch(\ell)$ because if we assume that $\beta \in Ch(\ell)$ then $\ell < \beta$ from Lemma 1, and thus the contradiction is derived. It immediately follows from Lemma 4 that $\mathcal{P}^r(S)$ is connected. Since $\beta < \ell$ again, we have $w(\mathcal{P}^r(S)) = w(S) - \ell + \beta < w(S)$. \square

From Lemma 6, it is natural to have \mathcal{I}^r as the sub-initial k -subtree of \mathcal{S}^r .

Example 2. *In Fig. 3.1, we observe that subtree S_6 is the parent of S_7 of type I since the maximum leaf is $\ell = 8$ and the minimum border vertex is $\beta = 3$, where $Lv(S_7) = \{2, 8\}$ and $Bd(S_7) = \{3, 4, 9, 10, 11, 12\}$.*

3.2.4 A inter-family tree for serial k -subtrees

To enumerate the whole \mathcal{S} , it is sufficient to compute the r -rooted serial k -subtree \mathcal{I}^r for each possible vertex r in T , and then to enumerate \mathcal{S}^r starting from \mathcal{I}^r . We see, however, that this approach is difficult to implement in constant delay because it is impossible to compute \mathcal{I}^r from scratch in the constant time.

To overcome this difficulty, we define the family tree \mathcal{F}^* . Then, we traverse between \mathcal{S}^r s using the parent function \mathcal{P}^* . The family tree is given by $\mathcal{F}^* = (\mathcal{S}^*, \mathcal{P}^*, \mathcal{I}^*)$, where \mathcal{S}^* is the collection of serial k -subtrees and pre-serial k -subtrees, $\mathcal{I}^* = \mathcal{I}_k^1(T)$ is the

unique serial k -subtree with root 1, and \mathcal{P}^* is the parent function from $\{\mathcal{I}^r \mid r \in V(T)\} \setminus \{\mathcal{I}^*\}$ to \mathcal{S} . The definition of a pre-serial k -subtree is given in Sect. 3.3.2. We define function \mathcal{P}^* as follows.

Definition 2 (the parent of k -subtree of type II). *Let S be any serial k -subtree other than \mathcal{I}^* . Then, the parent of S is the k -subtree*

$$\mathcal{P}^*(S) = (S \setminus \{\ell\}) \cup \{\beta\}$$

obtained from S by deleting the vertex $\ell = \max(Lv(S))$ and adding the vertex $\beta = pa(\text{root}(S))$. Then, we say that S is a type-II child of $\mathcal{P}^(S)$.*

Lemma 7. *If $S \in \mathcal{S}^* \setminus \{\mathcal{I}^*\}$, then $\mathcal{P}^*(S)$ is uniquely determined, and a well-defined k -subtree of T . Furthermore, $w(\mathcal{P}^*(S)) < w(S)$ holds.*

Proof. If S is not the initial k -subtree \mathcal{I}^* , β is always defined. Since ℓ is a leaf in S , $S' = (S \setminus \{\ell\})$ is obviously connected. Since β is adjacent to $\text{root}(S)$, clearly, $\mathcal{P}^*(S) = (S' \cup \{\beta\})$ is also connected. Since $\beta < v$ for any vertex v in S , $w(\mathcal{P}^*(S)) = w(S) - \ell + \beta < w(S)$ holds. \square

Example 3. *In Fig. 3.1, we observe that subtree S_8 is the parent of S_9 of type II since the maximum leaf is $\ell = 10$ and the parent of the root is $\beta = 1$, where $Lv(S_9) = \{9, 10\}$ and $Bd(S_9) = \{11, 12\}$.*

3.2.5 Putting them together

Recall that $\mathcal{S} = \mathcal{S}^1 \uplus \dots \uplus \mathcal{S}^n$. Let \mathcal{P}^r and \mathcal{P}^* be the parent functions for non-serial trees for every vertex r in T and serial trees defined in Sect. 3.2.3 and Sect. 3.2.4, respectively. Now, we define the *master family tree* \mathcal{F} for the family \mathcal{S} of all k -subtrees in T by

$$\mathcal{F} = (\mathcal{S}, \mathcal{P}, \mathcal{I}),$$

where $\mathcal{P} : \mathcal{S} \setminus \mathcal{I} \rightarrow \mathcal{S}$ is the disjoint union $\mathcal{P}^* \uplus \biguplus_{r \in V(T)} \mathcal{P}^r$, and $\mathcal{I} = \mathcal{I}^* = \mathcal{I}_k^1(T)$ is the initial k -subtree for T . By definition, $w(\mathcal{I}) = \frac{1}{2}k(k+1)$. Furthermore, it is not hard to see that $w(S) \geq w(\mathcal{I})$ for any k -subtree $S \subseteq V$.

Theorem 1. *The master family tree \mathcal{F} forms a spanning tree over \mathcal{S} .*

Proof. Suppose that starting from any $S \in \mathcal{S}$, we are to repeatedly apply the parent function \mathcal{P} to S . Then, we have a sequence of k -subtrees $S_0(= S), S_1, \dots, S_i, \dots$, where $i \geq 0$. From Lemma 6 and Lemma 7, the corresponding properly decreasing sequence of $w(S_0) > w(S_1) > \dots > w(S_i) > \dots$ has at most finite length since $w(S_i) \geq 0$. Since any subtree other than the initial k -subtree \mathcal{I} has the unique parent, the above sequence of k -subtrees eventually reaches \mathcal{I} in finite time. \square

From Theorem 1 above, we can easily show that all k -subtrees in \mathcal{S} can be enumerated in polynomial delay and polynomial space by a backtracking algorithm that traverses the family tree \mathcal{F} starting from the root \mathcal{I} .

Example 4. *In Fig. 3.1, $\mathcal{F}_k(T_1)$ is a spanning tree on $\mathcal{S}_k(T_1)$ rooted at $\mathcal{I}_k(T_1) = S_1$ for $k = 4$. Then, we can enumerate all 4-subtrees by traversing $\mathcal{F}^{(4)}(T_1)$.*

3.3 The Constant Delay Enumeration Algorithm

In this section, we present an efficient backtracking algorithm that enumerates all k -subtrees of an input rooted tree T in $O(1)$ delay using $O(n)$ preprocessing. The remaining task is to invert the reverse edges in \mathcal{P} to compute the children from a given parent. We describe this process according to the types of a child S .

3.3.1 Generation of non-serial k -subtrees

We first consider the case that a child S is non-serial (*type I*). In our algorithm, we keep these vertices as pointers to vertices in the implementation.

Definition 3. We define the candidate sets $DelList(R)$ and $AddList(R)$ for deleting vertices and adding vertices, respectively, as follows:

$$DelList(R) = \{\ell \in Lv(R) \mid \ell < \min(Bd(R))\},$$

$$AddList(R) = \{\beta \in Bd(R) \mid \beta > \max(Lv(R))\}.$$

Definition 4 (child generation of type I). Given a r -rooted k -subtree R in T , we define the k -subtree

$$Child_r(R, \ell, \beta) = (R \setminus \{\ell\}) \cup \{\beta\}$$

for any $\ell \in DelList(R)$, and for any $\beta \in AddList(R)$ such that β is not a child of ℓ .

Lemma 8 (update of lists). Let R be any r -rooted k -subtree and $S = Child_r(R, \ell, \beta)$ be defined for a leaf $\ell \in DelList(R)$ is removed from R and a border vertex $\beta \in AddList(R)$ is added to R . Then, the following (i) and (ii) hold.

(i) The leaf ℓ becomes the minimum border vertex in S .

(ii) The border vertex β becomes the maximum leaf in S .

Proof. From Lemma 2, we have that $\ell < \min(Bd(S)) < \max(Lv(S)) < \beta$. Hence, the conditions (i) and (ii) immediately follows. \square

The above lemma describes what happens when we apply $Child_r$ to R . Now, we show the correctness of $Child_r$ as follows.

Theorem 2 (correctness of $Child_r$). Let R and S be any r -rooted k -subtree of T , and S be non-serial. Then, (1) $R = \mathcal{P}^r(S)$ if and only if (2) $S = Child_r(R, \ell, \beta)$ for (i) some $\ell \in DelList(R)$ and (ii) some $\beta \in AddList(R)$ such that $\beta \notin Ch(\ell)$.

Proof. Firstly, we can easily obtain a statement that $Child_r(R, \ell, \beta)$ is non-serial from Lemma 2 and Lemma 4. (1) \Rightarrow (2): Suppose that $R = \mathcal{P}^r(S)$. Then, R is obtained

from S by removing $\ell_* = \max(Lv(S))$ and adding $\beta_* = \min(Bd(S))$. From Lemma 2, $\beta_* < \ell_*$. From Lemma 4, $\beta_* \notin Ch(\ell_*)$. Furthermore, any vertex $v \in S \setminus \{\ell_*\}$ is smaller than ℓ_* and any vertex $u \in (Bd(S) \setminus \{\beta_*\}) \cup Ch(\beta_*)$ is larger than β_* . Then, we see that $\max(Lv(R)) < \ell_*$ and $\beta_* < \min(Bd(R))$. If we put $\beta = \ell_*$ and $\ell = \beta_*$, then we can show that β and ℓ are a border vertex and a leaf in R , respectively, that satisfy the pre-condition of Child_r in Def. 4. Therefore, we can apply $\text{Child}_r(R, \ell, \beta)$, and then, we obtain the new child from R by removing $\ell = \beta_*$ from R and adding $\beta = \ell_*$ to R . From Lemma 5, the child is identical to the original subtree S . (2) \Rightarrow (1): In this direction, we suppose that $S = \text{Child}_r(R, \ell, \beta)$ for some $\ell \in Lv(R)$ and $\beta \in Bd(R)$ satisfying the conditions (i) and (ii). Then, S is obtained from R by removing ℓ from and adding β to R . From Lemma 8, ℓ becomes $\min(Bd(S))$ and β becomes $\max(Lv(S))$. Thus, if we put $\beta_* = \ell$ and $\ell_* = \beta$, then β_* and ℓ_* satisfies the pre-condition of \mathcal{P}^r in Def. 1. By applying \mathcal{P}^r to S , we easily see that $(S \setminus \{\ell_*\}) \cup \{\beta_*\} = R$. Hence, the theorem holds. \square

3.3.2 Generation of serial k -subtrees

Next, we consider the special case to generate a serial k -subtree as a child k -subtree S of a given parent k -subtree (*type II*). A k -subtree R is a *pre-serial k -subtree* if (i) $\text{root}(R)$ has only one child v such that $|T(v)| \geq k$, and (ii) v satisfies that $R \setminus \{\text{root}(R)\}$ is a serial $(k - 1)$ -subtree of T with root v .

Lemma 9. *R is a pre-serial k -subtree of T if and only if $\text{root}(R)$ has a single child v and the equality $\max(Lv(R)) = v + k - 2$ holds. Furthermore, we can check this condition in constant time.*

Proof. The result follows from that a pre-serial k -subtree is obtained from a serial $(k - 1)$ -subtree by attaching the new root as the parent of its root. \square

Definition 5 (child generation of type II). *For any pre-serial k -subtree R rooted at r , we define*

$$\text{Child}_*(R) = (R \setminus \{r\}) \cup \{\min(\text{Bd}(R \setminus \{r\}))\}.$$

Theorem 3 (correctness of Child_*). *Let R and S be any k -subtrees of T . Then, the following (i) and (ii) hold:*

(i) *If R is pre-serial, then $S = \text{Child}_*(R)$ implies $R = \mathcal{P}^*(S)$.*

(ii) *If S is serial and S is not \mathcal{I} , then $R = \mathcal{P}^*(S)$ implies $S = \text{Child}_*(R)$.*

Proof. Suppose that r is the root of R and $R' = R \setminus \{r\}$. From Lemma 2 and Lemma 9, if R is a pre-serial k -subtree, then we have $\min(\text{Bd}(R)) = \max(R) + 1$ and $\text{Child}_*(R)$ is serial. (i) Suppose that $S = \text{Child}_*(R)$ with deleting r and adding $\min(\text{Bd}(R'))$. Since r is the parent vertex of $\text{root}(S)$ and $\min(\text{Bd}(R'))$ is the largest vertex in S , application of \mathcal{P}^* to S yields R . (ii) Suppose that R is obtained from S by \mathcal{P}^* with adding the parent r' of $\text{root}(S)$ and deleting $\beta = \max(S)$. We can easily see $r' = r$. Since S is serial, we have $\max(R) = \max(S) - 1 = \beta - 1$ and then $\min(\text{Bd}(R')) = \max(R) + 1 = (\beta - 1) + 1 = \beta$, where $R' = R \setminus \{r\}$. Thus, we obtain S if we apply Child_* to R by deleting r and adding $\min(\text{Bd}(R'))$. This completes the proof. \square

3.3.3 The proposed algorithm

In Algorithm 1, we present the main procedure **EnumSubTrees** and the subprocedure **RecSubTree** that enumerates all k -subtrees in an input rooted tree T of size n in constant delay. Starting from \mathcal{I} , **RecSubTree** recursively computes all child k -subtrees from its parent by the child generation method in the previous subsections.

The subprocedure **RecSubTree** maintains the lists of vertices **AddList**(R) and **DelList**(R) so that it can efficiently find a pair of vertices ℓ in **DelList**(R) and β in **AddList**(R) at

Algorithm 1: Constant delay enumeration for all k -subtrees in a tree.

```

1 Procedure EnumSubTrees( $T, k$ )
    Input           : An input rooted tree  $T$  of size  $n$ , and size  $k$  of subtrees
                      ( $1 \leq k \leq n$ )
    Output          : All  $k$ -subtrees in  $T$ 
2   Global variable: the working stack  $W$ ;
3    $W = \emptyset$ ;
4   Number the vertices of  $T$  by the DFS-numbering;
5   Compute the initial  $k$ -subtree  $\mathcal{I}$  of the input rooted tree  $T$ ;
6   Update the related lists and pointers;
7   RecSubTree( $\mathcal{I}, T, k$ );
8 Subprocedure RecSubTree( $S, T, k$ )
    Input           : A  $r$ -rooted  $k$ -subtree  $S$ , an input rooted tree  $T$ , and size  $k$ 
                      of subtrees
    Output          :  $S$ 
9   Output  $S$ ;
10  foreach  $\ell \in DelList(S)$  do                                     // See Sect. 3.3.1
11      foreach  $\beta \in AddList(S)$  such that  $\beta \notin Ch(\ell)$  do
12           $S \leftarrow Child_r(S, \ell, \beta)$  by calling  $Update_r(\mathcal{A}, \ell, \beta)$ ;
13          RecSubTree( $S, T, k$ );
14           $S \leftarrow \mathcal{P}^r(S)$  by calling  $Restore_r(\mathcal{A})$ ;
15  if  $S$  is a  $k$ -pre-serial tree then                                // See Sect. 3.3.2
16       $S \leftarrow Child_*(S)$  by updating the related lists and pointers;
17      RecSubTree( $S, T, k$ );
18       $S \leftarrow \mathcal{P}^*(S)$  by restoring the related lists and pointers;

```

lines 10 and 11, respectively, when it generates a child of type I by calling Child_r satisfying the conditions of Def. 4. When it backtracks to the parent, we restore the update by performing the same set of operations in the reverse order. For the generation a child of type II by Child_* , we perform similar maintenance. For keeping the present values of ℓ and β , we use the *working stack* W .

For all parent R , we represent two lists $\text{AddList}(R)$ and $\text{DelList}(R)$ by the data structure \mathcal{A} with the following information:

- Doubly linked lists of vertices $Lv(R)$ and $Bd(R)$, which consist of all leaves and all border vertices of R , respectively. These lists are implemented by attaching two pointers prev_* and next_* to each vertex v of T in addition to the standard pointers for the leftmost and rightmost children, and next sibling [30].
- Two pointers $\hat{\ell} = \max(Lv(R))$ and $\hat{\beta} = \min(Bd(R))$ to vertices in $Lv(R)$ and $Bd(R)$, respectively.

The following operations $B2L(v)$ and $L2B(v)$ are fundamental to the maintaining of the data structure \mathcal{A} , where v is a vertex in T . B_0 , B_1 , L_0 , and L_1 are possibly empty sequences of vertices. We assume that vertices in the lists are sorted by the increasing order of DFS-numbering. Here, “ \circ ” indicates concatenating two lists.

- $B2L(v)$: Move a vertex v in $Bd(R)$ to $Lv(R)$. To implement this operation, we rewrite (i) the present list $Bd(R) = B_0 \circ \langle v \rangle \circ B_1$ to the new list $Bd(R) = B_0 \circ Ch(v) \circ B_1$, and (ii) $Lv(R) = L_0 \circ \alpha(v) \circ L_1$ to $Lv(R) = L_0 \circ \langle v \rangle \circ L_1$.
- $L2B(v)$: Move a vertex v in $Lv(R)$ to $Bd(R)$. To implement this, we rewrite (i) $Bd(R) = B_0 \circ Ch(v) \circ B_1$ to $Bd(R) = B_0 \circ \langle v \rangle \circ B_1$, and (ii) $Lv(R) = L_0 \circ \langle v \rangle \circ L_1$ to $Lv(R) = L_0 \circ \alpha(v) \circ L_1$.

In the above description, $\alpha(v)$ denotes the singleton sequence consisting only of $\text{pa}(v)$ if v has no sibling in R , and an empty sequence otherwise. We can show that two operations $B2L$ and $L2B$ are the inverse operations each other.

Using operations $B2L$ and $L2B$, we show the procedure for maintaining $\text{AddList}(R)$ and $\text{DelList}(R)$ at generation of a child of type I by Child_r in Algorithm 2, and at generation of a child of type II by Child_* in Algorithm 3.

Lemma 10 (time complexity of update). *Assuming the above representation, a data structure for the above lists and pointers can be implemented to run in $O(1)$ worst case time per update using $O(n)$ time preprocessing on RAM.*

Proof. Initialization of the data structure is done in $O(n)$ time by once traversing an input rooted tree T . At each request for update, we dynamically redirect pointers prev_* and next_* when a single vertex or a vertex list is deleted or added to R to maintain the values of $Lv(S)$ and $Bd(S)$ according to Algorithm 2. We can use a similar procedure to maintain lists and pointers in the case of Child_* according to Algorithm 3. Under this assumption, these operations can be implemented in the claimed complexity. \square

We have the main theorem of this chapter. This theorem shows we can enumerate all k -subtrees in an input rooted tree in constant delay.

Theorem 4. *Given an input rooted tree T of size n , and a positive integer $k \geq 1$, algorithm 1 solves the k -subtree enumeration problem in constant worst-case time per subtree using $O(n)$ preprocessing and space.*

Proof. By the construction of RecSubTree in Algorithm 1, we observe that each iteration of recursive call generates at least one solution. To achieve constant delay enumeration, we need a bit care to represent subtrees and to perform recursive call. From Lemma 10, each call performs constant number of update operations when it expands the current subtree to descendants. Therefore the remaining thing is to estimate the book-keeping on backtrack. This is done as follows: When a recursive procedure call is made, we apply a constant number of operations on candidate lists and record them on a stack as in Lemma 10, and when the procedure comes back to the parent

subtree, we apply the inverse of the recorded operations on the lists in constant time as in Lemma 10 to reclaim the running state in constant time. To improve the $O(d)$ time output overhead with backtrack from vertex v of depth $d = O(n)$ to a shallow ancestor on the family tree, we use alternating output technique (see, e.g., Uno [143]) to reduce it to exactly $O(1)$ time per solution. Combining the above arguments, we proved the theorem. \square

This result improves on the straightforward application of Ferreira *et al.*'s algorithm [46] with $O(k)$ amortized time per subtree when an input is restricted to a tree.

3.4 Application to the Graph Motif Problem for Trees

We consider the restricted version of graph motif problem [44, 88], called *the k -graph motif problem in a tree*. Then, we present an adaptive algorithm for the problem whose running time is proportional to the number of k -subtrees.

Let $\mathcal{C} = \{1, \dots, \sigma\}$ ($\sigma \geq 1$) be a set of *colors*. A *multiset* X on \mathcal{C} is a collection of possibly duplicated colors in \mathcal{C} . Precisely, $f_X(c)$ denotes the *count* of the color c in X . The *size* of multiset X is the total count of the colors in X defined by $\|X\| = \sum_{c \in \mathcal{C}} f_X(c)$. For multisets X and Y on \mathcal{C} , We define $X \subseteq Y$ if $f_X(c) \leq f_Y(c)$ for every $c \in \mathcal{C}$, and $X = Y$ if $f_X(c) = f_Y(c)$ for every $c \in \mathcal{C}$.

Let k be any non-negative integer and $T = (V, E, \text{root}(T), \xi)$ be a vertex colored, rooted tree, where $\xi(v)$ is the color of a vertex v in V . For a subset S of vertices, we denote by $\xi(S)$ the multiset of colors appearing in S . The *k -graph motif problem in a tree* is the problem of, given a vertex colored, rooted tree T and a multiset X on \mathcal{C} with size $k = \|X\|$, called a *pattern*, to find a k -subtree $S \subseteq V$ such that $\xi(S) = X$. This problem is known to be NP-hard even if an input is restricted to trees [44].

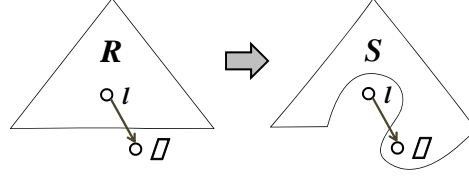


Figure 3.2: The bad case for Case 3 $\beta \in Ch(\ell)$ in the proof for the connectivity lemma (Lemma 4).

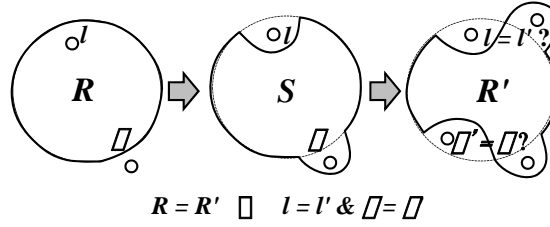


Figure 3.3: An idea of the proof for Lemma 5

Algorithm 2: Update of data structures $\text{DelList}(R)$ and $\text{AddList}(R)$ of type I

```

1 Procedure  $\text{Update}_r(\mathcal{A}, \ell, \beta)$ 
2   Push  $\hat{\ell}$  and  $\hat{\beta}$  in the working stack  $W$ ;
3    $L2B(\ell); B2L(\beta);$ 
4    $\hat{\ell} \leftarrow \beta; \hat{\beta} \leftarrow \ell;$ 
5 Procedure  $\text{Restore}_r(\mathcal{A})$ 
6    $\beta \leftarrow \hat{\ell}; \ell \leftarrow \hat{\beta};$ 
7    $L2B(\beta); B2L(\ell);$ 
8   Pop the values of  $\hat{\ell}$  and  $\hat{\beta}$  from  $W$ ;

```

In what follows, we denote by s the *number of all k -subtrees in a tree T* . From Theorem 4, we have the following result.

Theorem 5. *Let $k \geq 1$. Given an input rooted tree T of size n and a multiset X on \mathcal{C} with size k , the k -graph motif problem in a tree is solvable in $O(s + n + k)$ total time using $O(n + k)$ space.*

Proof. We give the algorithm **Adaptive** that solves the problem in the claimed complexity as follows: The algorithm uses a counter array g on \mathcal{C} , where the counter value $g[c]$ can take either a positive, zero, or negative integer. Given a pattern X , it first initializes the counter array by $g[c] \leftarrow -f_X(c)$ for each $c \in \mathcal{C}$ in $O(\sigma)$ time. It also initializes **EnumSubTrees** of Algorithm 1 in $O(n)$ time. Then, the algorithm enumerates all the k -subtrees S of T in $O(1)$ time per subtree. During the enumeration by **EnumSubTrees**, the algorithm increments (decrements, resp.) the counter $g[c]$ by one whenever a vertex labeled with color c is added to (is deleted from, resp.) S . This update of the counter can be done in constant time per subtree. When all counter values equal zero after the update, it outputs S as an output. The total running time of the algorithm is $O(s + n + k)$ time from Theorem 4. \square

We compare the time complexity of the algorithm **Adaptive** in the proof of Theorem 5 above to that of the straightforward algorithm, called **Naive** here, using exhaustive search for all k -subsets. **Naive** solves our graph motif problem as follows: First, **Naive** chooses k -subset S of T , and next, checks whether S is a k -subtree or not. Then, if S is a k -subtree, **Naive** compares the colors of S and input pattern X . **Naive** is done by applying the above procedure to all k -subsets of T .

From the upper bound in Lemma 3, **Adaptive** runs in time proportional to the actual number s of k -subtrees contained in an input rooted tree T , while **Naive** always requires $O(kn^k)$ time regardless of s . Therefore, we can say that our algorithm is indeed an adaptive algorithm so that it runs faster than **Naive** in the case that the number s

is much smaller than n^k .

Algorithm 3: Update of data structures $\text{DelList}(S)$ and $\text{AddList}(S)$ of type II

```

1 Procedure  $\text{Update}_*(\text{DelList}(S), \text{AddList}(S), r, \beta)$ 
   Input : Old lists  $\text{DelList}(S)$  and  $\text{AddList}(S)$ , and  $r = \text{root}(S)$  and
           deleted vertex  $\beta$ .
   Output : Update lists  $\text{DelList}(S)$  and  $\text{AddList}(S)$ .
2  $v \leftarrow \text{leftmostchild}_R(r)$ ; //  $v$  is the unique child of  $r$ .
3  $q \leftarrow \text{rightsinbling}_R(v)$ ;
4  $\text{DelList}(S) \leftarrow Lv(S)$ ;
5  $\text{AddList}(S) \leftarrow \text{AddList}(S) \setminus [q, \infty]$ ;
6  $\text{AddList}(S) \leftarrow \text{AddList}(S).\text{popfront}$ ;
7 if  $\beta \notin Lv(T)$  then // ‘‘ $\circ$ ’’ indicates concatenating two lists.
8    $\text{AddList}(S) \leftarrow Ch(\beta) \circ \text{AddList}(S)$ ;
9 return  $(\text{DelList}(S), \text{AddList}(S))$ ;

```

Chapter 4

Induced Tree Enumeration

In the 1970s, Read and Tarjan [113] studied a problem of enumerating spanning trees in the input graph. Their algorithm runs in $O(m + n + mN)$ time. Here, N is the number of solutions. Shioura, Tamura, and Uno [125] improved the complexity to $O(n + m + N)$ time. Tarjan [130] proposed an algorithm for enumerating all cycle in $O((|V| + |E|)(|\mathcal{C}(G)| + 1))$ time, where $\mathcal{C}(G)$ is all cycle in G . Birmelé *et al.* [13] improved the complexity to in $O(m + \sum_{c \in \mathcal{C}(G)} |c|)$ total time. They also presented an enumeration algorithm for all st-paths in the input graph G in $O(m + \sum_{\pi \in \mathcal{P}_{st}(G)} |\pi|)$ total time, where $\mathcal{P}_{st}(G)$ is all st-paths in G . Ferreira *et al.* [46] proposed an enumeration algorithm that enumerating all subtree having exactly k edges in G in $O(kN)$ time. Wasa *et al.* [152] presented an improved version of Ferreira *et al.*'s problem in constant time delay when the input is a tree. As we see, speed up of enumeration algorithms have been intensively studied in long history.

Compared to these studies, induced subgraph enumerations have not been studied well. Avis and Fukuda [7] considered the connected induced subgraph enumeration problem. Their algorithm is based on the reverse search, and runs in $O(mnN)$ time. Uno [140] proposed an enumeration algorithm for enumerating all chordless path connecting the given vertices s and t and all chordless cycle in $O((m + n)N)$ time. Ferreira

et al. [47] also proposed an enumeration algorithm for this problem. Their algorithm runs in $\tilde{O}(|n|)$ time per chordless cycle. Their algorithm also enumerates all st-chordless paths with the same complexity.

In this chapter, we address the problem of enumerating all induced trees in the given graph, where an induced tree is a connected induced subgraph that has no cycle. Assume that the set of vertices in an induced tree is S . Then, $V \setminus S$ is a feedback vertex set of G . Feedback vertices are also fundamental graph objects and their enumeration problem is equivalent to that of induced trees. If the input graph G is a tree, the connected induced subgraph of G is a subtree. In Chap. 3 we showed that the induced tree enumeration problem can be solved in constant time delay when the input graph is a tree. Tree is a simple graph class, so we are motivated whether we can do better in more general graph classes with non-trivial algorithms.

As a main result, we propose an algorithm for the k -degenerate graph case. The algorithm runs in $O(k)$ time per solution, after $(|V| + |E|)$ preprocessing time. The algorithm starts from the empty subgraph, and adds a vertex recursively to enlarge the induced tree. The vertex to be added has to be adjacent to the current induced tree, and has not to make a cycle. By using the degeneracy, we efficiently maintain the addible vertices, and the time complexity is bounded by a sophisticated amortized analysis. Real world graphs usually have small degeneracies, or only few vertex removals result small degeneracies, the algorithm is expected to be efficient in practice. Compared to other graph classes, this is a strong point of k -degenerate graphs. There have been not so many studies on the use of the degeneracy for enumeration algorithm, and thus our approach introduces one of new way of developing practically efficient and theoretically supported algorithms.

4.1 Preliminaries

Let $G = (V(G), E(G))$ be a graph and S be a subset of $V(G)$. We say that the induced graph $G[S]$ induced in S is an *induced tree* if $G[S]$ forms a tree (see Fig. 4.1). In what follows, since $G[S]$ is uniquely determined by S , we identify S with $G[S]$ if no confusion. For simplicity, if it is clear from context, we omit G from $V(G)$ and $E(G)$. Next, we define the enumeration problem considered in this chapter.

Problem 2 (induced tree enumeration). *Given an input graph G , enumerate all induced trees in G without duplicates.*

4.1.1 K -degenerate graphs

A graph G is *k -degenerate* [92] if any its induced subgraph of G has a vertex whose degree is less than or equal to k . The *degeneracy* of G is defined as the smallest k satisfying the definition of k -degenerate graphs. Examples of graph classes with constant degeneracy include trees, grid graphs, outerplanar graphs, and planer graphs, thus degenerate graph is a large class of sparse graphs. These degeneracy are 1, 2, 2, and 5, respectively.

From the definition of k -degeneracy, we obtain a vertex sequence $(u_1, \dots, u_{|V|})$ satisfying the condition

$$\forall 1 \leq i \leq |V|, |\{u_j \in N(u_i) \mid i < j \leq |V|\}| \leq k. \quad (\star)$$

This condition (\star) implies that there exists an *ordering* among vertices of G such that for any vertex u , the number of vertices adjacent to u larger than it is at most k . Hereafter we assume that the vertices are indexed in this ordering. We say $u < v$ ($u > v$, respectively) if the index of u is smaller than v (u is larger than v , respectively) with respect to this ordering. In Fig. 4.2, we show an example of the ordering satisfying (\star) .

Matula and Beck [99] proposed an algorithm for obtaining the degeneracy of G and the ordering satisfying (\star) . By iteratively choosing the smallest degree vertex and removing it from G , their algorithm finds such an ordering in $O(|V| + |E|)$ time.

4.2 Basic Binary Partition Algorithm

4.2.1 Candidate sets and forbidden sets

Let S be an induced tree of G . We define the *adjacency* of a vertex $u \in V$ to S as $\text{adj}(S, u) = |S \cap N(u)|$, that is, $\text{adj}(S, u)$ is the number of vertices of S adjacent to u .

Lemma 11. *Let S be any induced tree in G and u be any vertex $V \setminus S$. $S \cup \{u\}$ is an induced tree if and only if $\text{adj}(S, u) = 1$.*

Proof. If $\text{adj}(S, u) > 1$, u is adjacent to two vertices v and w of S . Since S has a path π connecting v and w , the addition of u yields a cycle in $S \cup \{u\}$. If $\text{adj}(S, u) = 0$, $S \cup \{u\}$ is disconnected. If $\text{adj}(S, u) = 1$, $S \cup \{u\}$ is connected. Since the degree of u in $G[S \cup \{u\}]$ is one, u is not included in a cycle. Thus, $G[S \cup \{u\}]$ does not contain a cycle. \square

In each iteration, we maintain the *forbidden set* X as the vertex set such that any vertex u in X satisfies either u belongs to S , $S \cup \{u\}$ includes a cycle, or u is forbidden to include in the solution by some ancestor iterations of the iteration. We also maintain the *candidate set* $CAND$ as the set of vertices whose additions yield induced trees and are not included in X . We maintain $CAND$ and X for efficient computation. From Lemma 11, they are disjoint, and for any vertex u , if $\text{adj}(S, u) > 0$, u belongs to either $CAND$ or X .

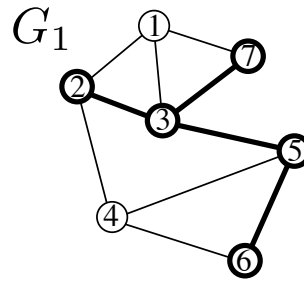


Figure 4.1: An induced subtree S_1 in G_1 . In the figure, bolded vertices and edges represent vertices and edges in S_1 . S_1 consists of $\{2, 3, 5, 6, 7\}$. S_1 is an induced subtree in G_1 since S_1 is connected and acyclic.

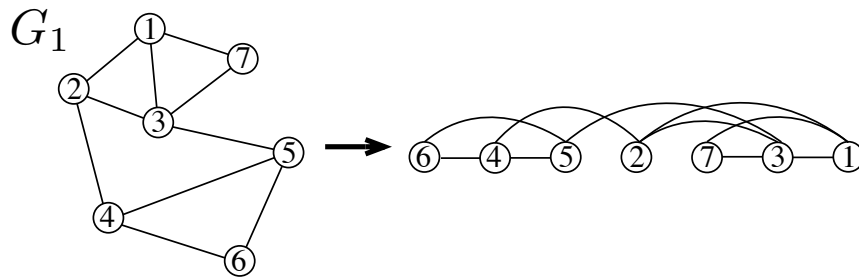


Figure 4.2: An example of an ordering of $G_1 = (V_1, E_1)$. In the right graph, vertices are sorted by the ordering that satisfies (\star) .

4.2.2 Basic binary partition

Our algorithm starts from the empty induced tree $S = \emptyset$. In each iteration given an induced tree S , we remove a vertex u from $CAND$, and partition the problem into two; enumeration of all induced trees including $S \cup \{u\}$, and those including S but not including u . We recursively do this partition until there is no vertex in $CAND$. The former can be solved by a recursive call with setting S to $S \cup \{u\}$. The latter is solved by a recursive call with setting X to $X \cup \{u\}$. In this way, we can enumerate all induced trees. We present the main routine **ISE** of our algorithm in Algorithm 4. We show how to update candidate sets and forbidden sets in the next two lemmas.

Lemma 12. *For an induced tree S and a vertex $u \in CAND$, when we add u to S and remove u from $CAND$, $CAND$ changes to*

$$(CAND \setminus N(u)) \cup (N(u) \setminus (CAND \cup X)).$$

Proof. Any vertex in $CAND$ other than $N(u)$ remains in $CAND$ after the addition of u to S since the adjacencies of the vertices do not change. If vertices in $N(u) \cap (CAND \cup X)$ are added to $S \cup \{u\}$, then they are in S , they are forbidden to be added to S and its descendants, or they make cycles since they are adjacent to u and other vertices in S . The adjacency of any vertex in $N(u) \setminus (CAND \cup X)$ is zero for S , and one for $S \cup \{u\}$. Any vertex $v \notin S$ satisfying $\text{adj}(S \cup \{u\}, v) = 1$ is either in $N(u)$ or $CAND$. Thus, the statement holds. \square

Lemma 13. *For an induced tree S and a vertex $u \in CAND$, when we add u to S and remove u from $CAND$, X changes to*

$$X \cup \{u\} \cup (CAND \cap N(u)).$$

Proof. Any vertex $v \in X$ remains in X for $S \cup \{u\}$, since $\text{adj}(S \cup \{u\}, v) \geq \text{adj}(S, v)$

always holds. From the definition of the forbidden set, u is in X for $S \cup \{u\}$. Further, any vertex v in $CAND \cap N(u)$ makes cycles when they are added to $S \cup \{u\}$, since $\text{adj}(S \cup \{u\}, v) \geq 2$ holds. By adding u to S , no other vertex is forbidden to be added, thus the statement holds. \square

Theorem 6. *Algorithm ISE enumerates all induced trees in the input graph $G = (V, E)$ without duplicates.*

Proof. From Lemma 12 and Lemma 13, Algorithm ISE correctly updates the candidate set and the forbidden set. Thus, from Lemma 11, the soundness and completeness of Algorithm ISE are satisfied. Hence, the statement holds. \square

4.3 Improved Binary Partition Algorithm

From Lemma 12 and Lemma 13, we can easily see that the computation time of updating the candidate set and the forbidden set is $O(d_G(u))$ by checking all vertices adjacent to u . However, in this way, we must check some vertices again and again. Specifically, let us assume u and v are consecutively added to S , and $w \notin S$ is adjacent to u , v , and another vertex in S . When we add u to S , we check whether we can add w to the candidate set of $S \cup \{u\}$. After generating $S \cup \{u\}$, we check w again when we add v to $S \cup \{u\}$. In order to avoid this redundant checking, we improve the way of updating the candidate set and the forbidden set by using the following set.

Definition 6. *Suppose that u is a vertex of CAND for an induced tree of G . We define a set $\Gamma(u, X)$ as follows:*

$$\Gamma(u, X) = \{v \in N(u) \mid v \notin X, v < u\}.$$

Lemma 14. *Let S be an induced tree of G , u be the smallest in the candidate set $CAND$ of S , and X be the forbidden set of S . Then, the following formula holds:*

$$N(u) \setminus (CAND \cup X) = (N_\ell(u) \setminus (CAND \cup X)) \cup \Gamma(u, X),$$

where $N_\ell(u) = \{v \in N(u) \mid u < v\}$.

Proof. Let Z be the set of vertices larger than u . Since u is the smallest vertex in $CAND$, $(N(u) \setminus (CAND \cup X)) \cap Z = (N_\ell(u) \setminus (CAND \cup X))$. From the definition of $\Gamma(u, X)$ and u is the smallest in $CAND$, $(N(u) \setminus (CAND \cup X)) \cap (V \setminus Z) = N_s(u) \setminus (CAND \cup X) = (N_s(u) \setminus CAND) \cap (N_s(u) \setminus X) = \Gamma(u, X)$, where $N_s(u) = \{v \in N(u) \mid v < u\}$. This concludes the lemma. \square

In what follows, we implement $CAND$, X , and Γ by doubly linked lists. Thanks to the doubly linked list, the cost for a removal and the recover of the removed element can be done in constant time, and the merge of two sets can be done in linear time of the sum of their sizes. In each iteration, we keep vertices of each list sorted in the ordering that satisfies (\star) .

Lemma 15. *When we add a vertex u to X , the update of $\Gamma(v, X)$ for all vertices v is done in $O(k)$ time.*

Proof. To update, it is suffice to remove u from $\Gamma(v, X)$ from all $v > u$. Thus, it takes $O(k)$ time. \square

Lemma 16. *Let S be an induced tree of G , u be the smallest in the candidate set $CAND$ of S , and X be the forbidden set of S . When we add u to S and remove u from $CAND$, the computation time of updating $CAND$ and X are $O(k + |\Gamma(u, X)|)$ and $O(k)$ time, respectively.*

Proof. Since u is the smallest vertex in $CAND$, $|\Delta| \leq k$, where $\Delta = |CAND \cap N(u)|$. Since vertices in $N(u)$ are sorted by the ordering, the computation time of Δ is $O(k)$.

Thus, adding vertices in Δ and u to X and removing Δ from $CAND$ are done in $O(k)$ time. From Lemma 14, since $|\{v \in N(u) \mid u < v\}| \leq k$, the computation time of adding these vertex to $CAND$ is $O(k + |\Gamma(u, X)|)$. Hence, the lemma holds. \square

In Fig. 4.3, we show the changes of between the candidate set of S and that of $S \cup \{u\}$ after adding u to S .

Theorem 7. *Let $G = (V, E)$ be the input graph and k is the degeneracy of G . Our algorithm enumerates all induced trees in G in $O(k)$ time per solution after $O(|V| + |E|)$ preprocessing time without duplicates using $O(|V| + |E|)$ space.*

Proof. Since the update of $CAND$ and X is correct, the correctness of the algorithm is obvious. (I) We discuss the time complexity of the preprocessing. First, our algorithm computes an ordering of vertices by Matula and Beck's algorithm [99] in $O(|V| + |E|)$ time. Next, our algorithm sorts vertices belonging to each adjacency list by using a bucket sort. Thus, the preprocessing time is $O(|V| + |E|)$.

(II) We consider an iteration inputting S , X , and $CAND$, and assume that $CAND'$ is the candidate set for $S \cup \{u\}$. Line 2 and line 3 run in $O(1)$ time. From Lemma 15, line 4 needs $O(k)$ time. From Lemma 16, since it is clear that $|\Gamma(u, X)| \leq |CAND'|$, our algorithm needs $O(k + |CAND'|)$ time for computing $CAND'$. The update of Γ 's is done in $O(k|CAND \cap N(u)|)$ time, from Lemma 15. We observe that for each vertex w such that $v \in CAND \cap N(u)$ is removed from $\Gamma(w, X)$, w is in $CAND$ of $S \cup \{v\}$, that will be generated by a descendant of this iteration. We charge the cost of constant time to remove v from $\Gamma(w, X)$ to the induced tree $S \cup \{v, w\}$. Then, we can see that $S \cup \{v, w\}$ is charged only from iterations inputting S , that divides the problem by u' such that $(u', v) \in E$, that is, the iteration generates $S \cup \{u'\}$. We consider the average amount of the charge over all induced trees of $S \cup \{v, w\}$, $v \in CAND$, and w is in $CAND$ of $S \cup \{v\}$. Since the number of pairs $\{u, v\} \subseteq CAND$ is at most $k|CAND|$, we can see the average charge is $O(k)$ for each $S \cup \{v, w\}$. Thus, in summary, we can see the

update time for Γ in an iteration is bounded by $O(k)$, on average. Thus, an iteration takes $O(k + |CAND'|)$ time on average. We observe that the sum of $|CAND'|$ over all iterations is no greater than the sum of $|CAND|$ over all induced trees, since $CAND'$ is the candidate set of $S \cup \{u\}$ and forbidden set $X \cup \{u\}$, and $S \cup \{u\}$ is generated only from S . Further, we can see that $S \cup \{u\}$ is generated only from S this iteration. Hence, thus the sum of $|CAND|$ over all induced trees is bounded by the number of induced trees. Therefore, the computation time for each iteration is bounded by $O(k)$ on average.

In a binary partition algorithm, each iteration at the leaf of the recursion outputs a solution, and each non-leaf iteration generates exactly two recursive calls. Thus, the number of iterations (recursive calls) of a binary partition algorithm is at most $2N$. Hence, the computation time per induced tree is $O(k)$. All sets the algorithm maintains are of size $O(|V| + |E|)$ in total.

We need a bit care to perform a recursive call. When a recursive call is made, we record the operations to prepare the parameters given to the recursive call on the memory. When the recursive call ends, we apply the inverse operations of the recorded operations to recover the variables such as $CAND$ and X . In this way, we can recover the variables from the updated ones without increasing the time complexity. Since no vertex is added or deleted from the same variable twice, the accumulated space for the recorded operations is bounded by $O(|V| + |E|)$. From the above arguments, our algorithm runs in $O(k)$ time per solution after $O(|V| + |E|)$ preprocessing time using $O(|V| + |E|)$ space. \square

Algorithm 4: Main routine ISE: Enumerating all induced trees in G

```

1 Procedure ISE( $G = (V, E), S, CAND, X$ )
2   If  $CAND = \emptyset$  then output  $S$ ; return;
3   Choose the smallest vertex  $u$  from  $CAND$  and remove  $u$  from  $CAND$ ;
4   ISE( $G, S, CAND, X \cup \{u\}$ );
5   ISE( $G, S \cup \{u\}, (CAND \setminus N(u)) \cup (N(u) \setminus CAND), X \cup \{u\} \cup (CAND \cap N(u))$ );

```

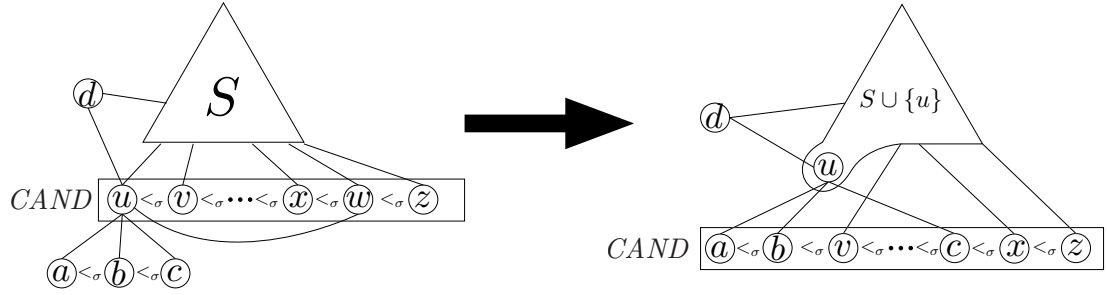


Figure 4.3: This figure shows the changes between candidate set $CAND$ by the addition of u to S . S is an induced tree and $\{u, v, \dots, x, w, z\}$ is the candidate set of S . Let assume that $a < b < u < c$ and $d < u$. Since d does not belongs to $\Gamma(u, X)$, d is skipped checking.

Chapter 5

Berge-Acyclic Subhypergraph Enumeration

In this chapter, we study the problem of enumerating all connected and acyclic sub-hypergraphs contained in an input hypergraph for the notions of acyclicity, called Berge-acyclicity [11], which is at the bottom of hierarchy of acyclicities given by Fagin [43]. Essentially, a hypergraph is a representation of any group relations, or any *set collection* consisting of groups of objects taken from the universe. For example, the followings are examples of such set collections: transaction databases, author groups in bibliographic data, co-citation networks in social networks, and interaction graphs for genes and proteins in bioinformatics [91]. In such networks, discovery of some classes of subsets, such as connected components, connected subtrees, cliques, quasi-cliques, and dense subgraphs have been extensively studied in the context of network mining [91, 126, 144].

We present efficient depth-first algorithms **BergeEnum** (Theorem 8) that finds all connected Berge-acyclic sub-hypergraphs contained in an input hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ without duplicates in $O(nm)$ delay and $O(N)$ words of space, where $n = |\mathcal{V}|$, $m = |\mathcal{E}|$, and $N = ||\mathcal{E}||$ are the numbers of vertices, the number of hyperedges, and the total

size of the hyperedges in \mathcal{H} . To achieve polynomial delay and space complexity, our algorithm searches for all solutions in the depth-first manner called on a tree-shaped search space called a *family tree* without using any extra memory for table-lookup. This search space is designed based on a characterization of Berge-acyclic sub-hypergraphs given by us, with which we proposed an efficient and complete pruning strategy.

Next, we present the faster version of the algorithm, called **FastBergeEnum**, using adaptive computation. The algorithm uses incremental computation of the maximum border set, and finds all connected Berge-acyclic sub-hypergraphs contained in $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ without duplicates in $O(||S|| + ||N(S)||)$ delay using $O(N)$ space and $O(N)$ preprocessing, where $||S|| = O(nm)$ is the total length of hyperedges in S and $||N(S)||$ is the sum of the lengths of hyperedges adjacent to S . This algorithm has the delay that depends only on the size $||S|| + ||N(S)||$ of a discovered subset S and its neighbors, and thus, it will be more efficient for the large inputs in the real world.

Related work

For the class of α -acyclic sub-hypergraphs [43], Hirata *et al.* [66] presented an efficient algorithm that finds one of the maximal connected and acyclic sub-hypergraphs in an input hypergraph in linear time in the total input size. Extending this work, Daigo and Hirata [31] presented a polynomial delay and space algorithm that finds all connected and acyclic sub-hypergraphs in an input hypergraph. For the class of Berge-acyclic sub-hypergraphs, Lovász [94] showed a polynomial time algorithm that finds one of the maximal connected and Berge-acyclic sub-hypergraphs in an input hypergraph.

As closely related work, Ferreira *et al.* [46] presented an efficient algorithm for finding all distinct subtrees of size k in an input graph in $O(k)$ time (time per solution) and space, and Wasa *et al.* [152] the improved version in constant delay when an input is a tree. However, their approaches cannot be directly applicable to our problem.

In the case that the maximum size of hyperedges, the *rank*, is restricted to two, the

problem coincides to the well-known spanning tree problem for undirected graphs. For the problem, Read and Tarjan [113] first presented an $O(ns)$ time and $O(n)$ space algorithm in 1960's, where n is the number of edges in G . Recently, Shioura, Tamura, and Uno [125] presented $O(n + s)$ time and $O(n)$ space algorithm. Unfortunately, it is not easy to extend the algorithms for spanning tree enumeration to subtree enumeration.

5.1 Preliminaries

In what follows, we fix an input hypergraph $\mathcal{H} = (\mathcal{V}(\mathcal{H}), \mathcal{E}(\mathcal{H}))$ consisting of n vertices and m hyperedges. For the analysis of the adoptive complexity of the algorithm in Sect. 5.3, which depends only on the solution \mathcal{S} , we define the *neighbor size* of \mathcal{S} by $\|N(\mathcal{S})\| = \sum_{e \in \mathcal{S}} |N(e)|$.

5.1.1 Berge-acyclicity

A *path* between hyperedges $e, f \in \mathcal{E}$ is a sequence $\pi = (e_1 = e, e_2, \dots, e_k = f)$ ($k \geq 1$) of hyperedges that satisfies the condition $e_i \cap e_{i+1} \neq \emptyset$ for every $1 \leq i \leq k - 1$. A subset \mathcal{E} is *connected* if any pair of hyperedges e and f has some path between them in \mathcal{E} . By definition, the empty set and singleton set of hyperedges are connected.

Example 5. In the hypergraph \mathcal{H}_1 in Fig. 5.1, both of the subsets $S_1 = \{1, 2, 3, 4\}$ and $S_2 = \{1, 2, 4, 5\}$ are connected. On the other hand, the subset $S_3 = \{1, 3, 5\}$ is not connected since there is no path between the edges 1 and 5, and also the edges 3 and 5.

Definition 7 ([11]). A Berge-cycle (or simply a cycle) in a hypergraph \mathcal{H} is a sequence $\pi = (e_1, x_1, \dots, e_k, x_k)$ ($k \geq 2$) that satisfies the following conditions (i)–(iii):

(i) e_1, \dots, e_k are mutually distinct hyperedges.

(ii) x_1, \dots, x_k are mutually distinct vertices.

(iii) For each $1 \leq i \leq k-1$, $x_i \in e_i \cap e_{i+1}$ holds, and $x_k \in e_k \cap e_1$ holds,

where k is called the length of the cycle π . Then, we say that the set $\{e_1, \dots, e_k\}$ of hyperedges forms a Berge-cycle.

We denote by $\mathcal{CA}(\mathcal{H})$ the class of all connected and Berge-acyclic sub-hypergraphs in an input hypergraph \mathcal{H} . Intuitively, a Berge-cycle is a path of length more than or equal to two that starts from some hyperedge and returns to the start. A hypergraph \mathcal{H} is Berge-acyclic if it contains no Berge-cycles. From the construction of minimum length cycle S_5 in Fig. 5.1, In the next lemma, we show a fundamental property of Berge-acyclicity.

Example 6. Let $\mathcal{V}_1 = \{p, q, r, \dots, x, y, z\}$ be a set of eleven vertices. In Fig. 5.1, we show an example of a hypergraph $\mathcal{H}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ on \mathcal{V}_1 . In \mathcal{H}_1 , the hyperedge set \mathcal{E}_1 consists of six hyperedges $e_1 = \{q, r\}$, $e_2 = \{r, s, t, u, v, w\}$, $e_3 = \{q, v, x\}$, $e_4 = \{s, y\}$, $e_5 = \{w, z\}$, and $e_6 = \{p, v, x\}$, where the hyperedge e_i , $1 \leq i \leq 6$, is represented by the index i .

Lemma 17 (Berge [11]). If two hyperedges e and f contain mutually distinct vertices x and y in common, i.e., $x, y \in e \cap f$, then they form a Berge-cycle.

Proof. Take a path $\pi = (e, x, f, y)$ as a Berge-cycle. □

Example 7. In the hypergraph \mathcal{H}_1 in Fig. 5.1, the hyperedge subset $S_4 = \{1, 2, 3\}$ forms a Berge-cycle $\pi_4 = (1, r, 2, v, 3, q)$ of length three. From Lemma 17, we also see that the pair of hyperedges $S_5 = \{3, 6\}$ forms a Berge-cycle $\pi_5 = (3, x, 6, v)$ of length two since hyperedges 3 and 6 share common vertices x and v .

By definition, the empty set and any singleton sets of hyperedges are Berge-acyclic. From the next lemma, Berge-acyclicity is closed under subsets.

Lemma 18. If a non-empty subset S is Berge-acyclic, then any subset S' of S is also Berge-acyclic.

Proof. If S' has a Berge-cycle C , then S also has a Berge-cycle that consists hyperedges in C . This contradicts the assumption. Thus, the statement holds. \square

From Lemma 17 above, we see that Berge-acyclicity has strong restriction compared to other notions of hypergraph acyclicities. Actually, there is a hierarchy of acyclicities for hypergraphs, called the *degrees of acyclicities* of Fagin [43], that consists of α -acyclicity, β -acyclicity, γ -acyclicity, and Berge-acyclicities. In this hierarchy, α -acyclicity is most general, while Berge-acyclicity is most restricted. Now, we state our enumeration problem.

Definition 8 (Connected and Berge-acyclic sub-hypergraph enumeration). *Given an input hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, the task is to find all connected, and Berge-acyclic sub-hypergraphs $S \subseteq \mathcal{E}$ in \mathcal{H} belonging to the class $\mathcal{CA}(\mathcal{H})$ without duplicates.*

Example 8. *Consider the hypergraph \mathcal{H}_1 in Fig. 5.1 again. Then, the hyperedge subset $S_1 = \{1, 2, 3, 4\}$ is not a connected and Berge-acyclic subset in $\mathcal{CA}(\mathcal{H}_1)$ because it is connected but has a Berge-cycle. On the other hand, the hyperedge subset $S_2 = \{1, 2, 4, 5\}$ is a connected and Berge-acyclic subset in $\mathcal{CA}(\mathcal{H}_1)$*

5.1.2 Other Definitions and Properties

Leaves and connection counts

Let $S \subseteq \mathcal{E}$ be a subset of hyperedges, or a sub-hypergraph of \mathcal{H} . A hyperedge e *connects* S if the intersection $e \cap V(S)$ is not empty. Any vertex x in the intersection is called a *connection point*. Then, the *connection count* of e relative to S is defined by $\text{cnt}(e, S) = |e \cap V(S)| \geq 0$. In the next section, we give a characterization of connected and Berge-acyclic sub-hypergraphs using the connection count.

A *leaf* of a subset S is a hyperedge $e \in S$ such that $\text{cnt}(e, S - e) = 1$, that is, e has a single connection point in S except itself. Clearly, the empty subset \emptyset has no leaf

at all, and any singleton $S = \{e\}$ has the hyperedge e as its only leaf. We denote by $Lv(S)$ the set of all leaves of S . Actually, $Lv(S) = \{e \in S \mid \text{cnt}(e, S \setminus \{e\}) = 1\}$.

Representation of hypergraphs

Let $x \in \mathcal{V}$ be a vertex, and $e, f \in \mathcal{E}$ be hyperedges. If $x \in e$ holds, then we say that either the vertex x is *contained in* a hyperedge e , or the hyperedge e is *incident to* a vertex x . We denote the set of all hyperedges in \mathcal{E} that is incident to the vertex x by $N(x) = \{f \in \mathcal{E} \mid x \in f\}$. Hyperedges f is *adjacent* to e or e is a *neighbor* of f if f overlaps e , that is, $f \cap e \neq \emptyset$ holds.

Using the incident relation, a hypergraph \mathcal{H} with n vertices and m hyperedges is represented in our algorithms, to be described later, as an $n \times m$ binary matrix $A = (a_{i,j}) \in \{0, 1\}^{m \times n}$, called the *incident matrix* of \mathcal{H} , in a standard way, where for every $1 \leq i \leq n$ and $1 \leq j \leq m$, $a_{i,j} = 1$ if and only if the i -th vertex x_i is contained in the j -th hyperedge e_j , that is, $x_i \in e_j$ ($1 \leq i \leq n, 1 \leq j \leq m$) holds. In Fig. 5.2, we show an example of an incident matrix. Actually, each row $1 \leq i \leq n$ represents the incident set $N(x_i)$ of x_i , and each column $1 \leq j \leq m$ represents the corresponding hyperedge e_j itself as a set of vertices.

Data structure

In our algorithms in Sect. 5.2 and Sect. 5.3, we use a dynamic data structure, denoted by \mathcal{D} , similar to the *DLX* (also known as “*Dancing Links*”) data structure by Knuth [83], for efficiently and dynamically maintaining a given subset of hyperedges of \mathcal{H} in the form of incident matrix. The difference of our structure from Knuth’s DLX is the use of the dynamic predecessor dictionaries (such as the hash table or the `map` collection of C++/STL or Java) [30].

Our data structure \mathcal{D} stores the incident matrix of a set collection $D \subseteq \mathcal{E}$ in linear words of space in $||D||$ supporting the following operations: (i) retrieval of a

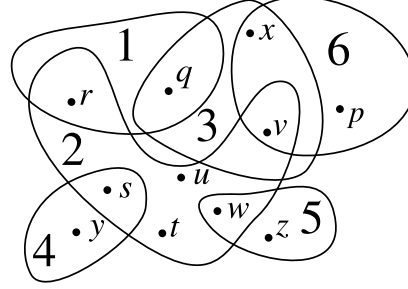


Figure 5.1: An example of a hypergraph $\mathcal{H}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ with the vertex set $\mathcal{V}_1 = \{p, q, r, \dots, x, y, z\}$ and the hyperedge set $\mathcal{E}_1 = \{1, 2, 3, 4, 5, 6\}$, where each point with a lowercase letter indicates a vertex and each region with a digit surrounded by a solid line indicates a hyperedge. The subset of hyperedges $S = \{1, 2, 4, 5\}$ forms a connected and Berge-acyclic sub-hypergraph of \mathcal{H}_1 to discover, while its super set $\{1, 2, 3, 4, 5\}$ is not acyclic.

$$A = \begin{matrix} & p & q & r & s & t & u & v & w & x & y & z \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} & \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix} \end{matrix}$$

Figure 5.2: The incident matrix A_1 of the hypergraph \mathcal{H}_1 in Fig. 5.1, where each row indicates a hyperedge and each column indicates a vertex.

hyperedge $e = e_i$ by an edge ID i , (ii) retrieval of the neighbor $N(x, D)$ by a vertex x , and (ii) insert/delete of elements to/from an edge or a neighbor set. Such a data structure can be implemented by linked lists and dynamic predecessor dictionary that allows to execute the above operations in sublinear time $t = f(k)$, where we have $f(k) = \log k$ if we use ordinary binary tree, and $f(k) = O(((\log \log k)^2 / \log \log \log k))$ for $k = \max \{n, m\}$ if we use the dynamic data structure of [9]. The details are omitted here.

5.2 The Basic Algorithm

In this section, we show that the basic version of our DFS enumeration algorithm **BergeEnum** that finds all connected and Berge-acyclic sub-hypergraphs in \mathcal{H} in polynomial delay and space. To devise efficient depth-first search algorithm, we need a systematic way to reduce the search for larger subsets to smaller subsets. The next lemma is essential to our algorithm.

Lemma 19. *Let $S' \subseteq \mathcal{E}$ is a subset such that $|S'| \geq 2$. If S' is connected and Berge-acyclic, then $\text{cnt}(e, S' - \{e\}) = 1$ holds for some $e \in S'$. Furthermore, $S = S' - \{e\}$ is connected and Berge-acyclic, too, and has size $|S| < |S'|$.*

Proof. We can show the lemma by induction on $|S'|$. If $|S'| = 2$, the claim is obvious since S' consists of two edges. Otherwise, assume that $|S'| > 2$. Since S' is connected, $\text{cnt}(e, S - e) \geq 1$ always holds for any $e \in S'$. Furthermore, if $\text{cnt}(e, S - e) \leq 1$ holds for some $e \in S'$, then we are done. Therefore, we assume that $\text{cnt}(e, S - e) \geq 2$ holds for any $e \in S'$. Consider this case. Then, we split S' by removing e , and consider the connected components S_1, \dots, S_k of $S' - e$, where $k \geq 1$. There are two cases. (i) If e connects to some S_i at at least two points, then $S_i \cup \{e\}$, and thus S' , immediately has a cycle, and we are done (ii) Otherwise, using induction hypothesis, we can show that there exists an edge f in some component, say S_1 , such that $\{e, f\} \cup R$ forms a cycle

for some $R \in \{\{e\}, S_1 - f, S_2, \dots, S_k\}$ (details are omitted), and we are done. Hence, by contradiction, the lemma follows. \square

From Lemma 19 above, starting from any connected and Berge-acyclic subhypergraph S with more than one edges, we can obtain a series of sub-hypergraphs $\mathcal{R} = S_0 = S \supset S_1 \supset \dots \supset S_\ell = \{e\}$ of length $\ell = |S| - 1 \geq 0$. Our DFS algorithm reverses this process by starting from any singleton set $\{e\}$, $e \in \mathcal{E}$, and by iteratively expanding the current subset $S \subseteq \mathcal{E}$ by adding new hyperedge $e \in \mathcal{E} \setminus S$ in a systematic manner using backtracking.

However, there is one problem with this approach. The above DFS search process may generate the same subset by exponentially many different paths. One easy way to avoid this duplication is to use table-lookup. When we discovered a new subset S , we lookup a hash table H to decide if $S \in H$. If so, we skip S , and otherwise, we output S and register it to H . This modification yields a polynomial delay, but exponential space enumeration algorithm.

We solve this problem by pruning of redundant path by careful design of the tree-shaped search space described as follows. Recall the previous key lemma, Lemma 19. In the lemma, the possible source of redundancy is more than one choice of a leaf $e \in S$ of S to delete. An idea to solve this is to restrict the deletion in reduction (and the addition in generation) to the *maximum* leaf of S . This ensures the reduction sequence $\mathcal{R} = S_0, \dots, S_\ell$ for S to be unique to each S . We call such a unique sequence the *maximum elimination sequence* for a sub-hypergraph S , and denote by $\mathcal{MES}(S)$.

Lemma 20. *$\mathcal{MES}(S)$ is the unique signature of each connected and Berge-acyclic sub-hypergraph $S \subseteq \mathcal{E}$.*

Proof. From Lemma 19, S always has an sequences obtained by recursively eliminating a leaf of S . Thus, we can always obtain $\mathcal{MES}(S)$. \square

From this lemma, we can generate S in the unique way by generating $\mathcal{MES}(S)$

instead. Now, we describe our algorithm. Firstly, we define the parent of a connected and Berge-acyclic sub hypergraph.

Definition 9. *Let S' be any connected and Berge-acyclic subhypergraph such that $|S'| \geq 2$. Then, the parent of S' is the set $\mathcal{P}(S') := S' - f$, where f is the leaf of S' such that $\text{cnt}(f, S' - f) = 1$ having the maximum edge ID among all leaves, that is, $f = \max(L(S'))$. This condition is called the maximum leaf condition. In this case, we call S' a child of $\mathcal{P}(S')$.*

Then, we define our tree-shaped search space called a *family tree*. The family tree for the class $\mathcal{CA}(\mathcal{H})$ of connected, and Berge-acyclic sub-hypergraphs of \mathcal{H} is a multi-rooted DAG $\mathcal{T} = (\mathcal{CA}(\mathcal{H}), \mathcal{P}, \mathcal{I}(\mathcal{H}))$, where

- $\mathcal{CA}(\mathcal{H})$ is the vertex set of \mathcal{T} that consists of all connected, and Berge-acyclic sub-hypergraphs in an input hypergraph \mathcal{H} .
- \mathcal{P} defines the set of reverse edges of \mathcal{T} that assign the parent $\mathcal{P}(S')$ to a child S' .
- $\mathcal{I}(\mathcal{H})$ is the set of all single subsets as the root nodes of \mathcal{T} .

In what follows, we write \mathcal{CA} and \mathcal{I} instead of $\mathcal{CA}(\mathcal{H})$ and \mathcal{I} , respectively. The next lemma says that the family tree is actually a tree-shaped search root.

Lemma 21. *For any input hypergraph \mathcal{H} , the family tree for \mathcal{CA} on \mathcal{H} is a spanning forest that contains all connected and Berge-acyclic subsets in \mathcal{CA} as its nodes.*

Proof. From Lemma 19, it immediately follows that $\mathcal{P}(S')$ is always connected, and Berge-acyclic, and has size strictly smaller than S' . Since each path of \mathcal{T} is a \mathcal{MES} for some element of \mathcal{CA} , \mathcal{T} is connected at some root in \mathcal{I} . On the other hand, since each reverse edge strictly reduces the size of S , \mathcal{T} contains no cycle. Hence, the lemma is proved. \square

In Algorithm 5, we show our basic DFS algorithm **BergeEnum** and its recursive subprocedure **BasicRec** that finds all connected, and Berge-acyclic sub-hypergraphs in \mathcal{H} in depth-first manner. This algorithm is a simple backtracking algorithm, working as follows. Starting from each singleton subset $\{e\}$ in \mathcal{I} , the algorithm searches the family tree \mathcal{T} for connected, and Berge-acyclic subsets by expanding the parent subset S by adding a new leaf f to obtain a child $S' = S \cup \{f\}$. In the expansion, it apply pruning for redundant subsets using the definition of a correct child based on the maximum leaf condition of the child. If expansion is no longer possible, it backtrack to the parent.

To compute the border set, we use the procedure **ComputeBorer** in Algorithm 6.

Lemma 22. *The algorithm **ComputeBorer** in Fig. 6 computes the border set of an hyperedge subset S in $O(nm)$ time using $O(n)$ additional space.*

Proof. From the definition, this algorithm correctly computes every borders. Line 5 needs $O(n)$ time to check all vertices in S . This computation needs $O(n)$ space. In addition, Line 5 are executed $O(m)$ time since Line 4. On the other hand, each vertex in an edge e in S is checked at most two times. Thus, **ComputeBorer** needs $O(nm)$ time. \square

We give the time and space complexity of the basic algorithm below.

Theorem 8 (main result). *The algorithm **BergeEnum** of Algorithm. 5 finds all connected Berge-acyclic sub-hypergraphs contained in an input hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ without duplicates in $O(nm)$ delay and $O(N)$ words of space, where $n = |\mathcal{V}|$, $m = |\mathcal{E}|$, and $N = ||\mathcal{E}||$ are the numbers of vertices and hyperedges, and the total size of the hyperedges in \mathcal{H} .*

Proof. Firstly, **BergeEnum** traverses the family tree \mathcal{T} for connected and Berge-acyclic subsets of \mathcal{H} , since **ComputeBorer** correctly computes the border of each connected and Berge-acyclic subset of \mathcal{H} from Lemma 22. This implies the correctness of the

Algorithm 5: A basic algorithm **BergeEnum** for enumerating all connected, Berge-acyclic sub-hypergraphs based on the reverse search

```

1 Procedure BergeEnum( $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ )
  |   Input           :  $\mathcal{H}$ : input hypergraph
2  |   BasicRec( $\emptyset, \mathcal{H}$ );
3 Subprocedure BasicRec( $S, \mathcal{H}$ )
  |   Input           :  $S$ : sub-hypergraph,  $\mathcal{H}$ : input hypergraph
4  |   Output  $S$ ;
5  |    $Border(S) \leftarrow \{f \in (\mathcal{E}(\mathcal{H}) \setminus S) \mid cnt(f, S) = 1\}$ ;
6  |   foreach  $f \in Border(S)$  do                                // Generation of children
7  |   |    $S' \leftarrow S \cup \{f\}$ ;
8  |   |   if  $f = \max Lv(S')$  then
9  |   |   |   BasicRec( $S', \mathcal{H}$ );

```

Algorithm 6: The algorithm for computing the border set of a sub-hypergraph

```

1 Procedure ComputeBorder( $S, \mathcal{V}, \mathcal{E}$ )
  |   Output           :  $Border(S) = \{f \in (\mathcal{E} \setminus S) \mid cnt(f, S) = 1\}$ 
2  |   Mark all vertices of  $\mathcal{V}(S)$ ;
3  |    $Border \leftarrow \emptyset$ ;
4  |   foreach  $e \in \mathcal{E}(S)$  do
5  |   |   Count the number  $cnt(e, S)$  of all marked vertices in  $e$ ;
6  |   |   if  $cnt(e, S) = 1$  then
7  |   |   |    $Border \leftarrow Border \cup \{e\}$ ;
8  |   return  $Border$ ;

```

algorithm is satisfied. Next, we consider the time complexity. From Line 4, each execution of **BasicRec** outputs one solution. Thus, the delay can be evaluated by the time complexity of **ComputeBorer** and the size of $Border(S) = O(m)$. From Lemma 22, it takes $O(nm)$ time. Since each execution in the loop of Line 6 takes $O(1)$ time. Thus, the delay is $O(nm + m) = O(nm)$ time. On the other hands, we have to mark every vertices in every edges for each execution of **ComputeBorer**, **BergeEnum** needs $O(N)$ space. Hence, the theorem holds. \square

From the theorem, we have the following corollary.

Corollary 9. *The class of all connected Berge-acyclic sub-hypergraphs contained in an input hypergraph \mathcal{H} can be enumerated in polynomial delay and polynomial space in the size of input \mathcal{H} .*

5.3 The Modified Algorithm

In this subsection, we show a modified version of our depth-first enumeration algorithm that finds all connected, Berge-acyclic sub-hypergraphs in an input hypergraph \mathcal{H} in $O(||S||)$ time using $O(N)$ space and preprocessing, where $N = ||\mathcal{E}(\mathcal{H})||$. This algorithm is *adaptive* since its time complexity only depends on the size of the discovered sub-hypergraph S , rather than the whole input. This adaptively is quite important in mining a large hypergraph.

The basic idea of our modified algorithm is incremental maintenance of the subset $MaxBorder(S)$ of hyperedge candidates to insert, called the maximal border hyperedges.

Definition 10. *The maximal border of a sub-hypergraph S is the set of hyperedges defined by:*

$$MaxBorder(S) = \{e \in \mathcal{E} \setminus S \mid cnt(e, S) = 1, e = \max L(S \cup \{e\})\},$$

that is, $MaxBorder(S)$ consists of all hyperedges e of \mathcal{H} satisfying the next conditions: (i) e is a border of S (i.e., $cnt(e, S) = 1$), and (ii) e is the maximum leaf of $S' = S \cup \{e\}$ among all leaves when it is added to S .

In Algorithm 8, we show our modified depth-first algorithm **FastBergeEnum** as well as its recursive subprocedure **FastRec** for enumerating all connected, Berge-acyclic subhypergraphs incrementally. By using $MaxBorder(S)$, in our depth-first enumeration algorithm **FastBergeEnum**, we can generate any children $S' = S \cup \{f\}$ from a parent S by just selecting any hyperedge $e \in MaxBorder(S)$ without testing the pruning condition for duplication because the condition is already included by the definition of the maximal border set. In other words, we are eager to make selection of border candidates and test for duplication at the same time in advance.

Therefore, it remains how to efficiently compute the maximal border set. Surprisingly, we can show that this is done in almost optimal time complexity in amortized analysis using a procedure similar to the α -acyclicity test by (Tarjan and Yannakakis [131]). The key to the algorithm is the following recurrence relation for the maximum and the second maximum leaves when we update a parent S by adding a new maximum border $f \in MaxBorder(S)$ to generate a children $S' = S \cup \{f\}$.

Lemma 23. *Let us denote by $max(S)$ and $2max(S)$ the maximum and the second maximum leaves of a parent set $S \subseteq \mathcal{E}$. Then, the maximum and the second maximum leaves $max(S')$ and $2max(S')$ of a child $S' = S \cup \{f\}$ satisfy the following recurrence:*

- If f connects $\ell_{\max} = maxLeaf(S)$:
 - If $f > 2max(S)$, then $max(S') \leftarrow f$ and $2max(S') \leftarrow max(S)$ hold.
 - Otherwise, $max(S') \leftarrow max(S)$ and $2max(S') \leftarrow 2max(S)$ hold.
- Otherwise:
 - If $f > max(S)$, then $max(S') \leftarrow f$ and $2max(S') \leftarrow max(S)$ hold.

– Otherwise, $\max(S') \leftarrow \max(S)$ and $2\max(S') \leftarrow 2\max(S)$.

Proof. In each case, the proof immediately follows from the case analysis using the definitions of \max , $2\max$, and the maximal border set. \square

From Lemma 23 above, we can update $\max(S)$ and $2\max(S)$ incrementally in constant time. Now, we show the algorithm **UpdateMaxBorder** in Algorithm 7 that incrementally updates the new border set $\text{MaxBorder}(S \cup \{f\})$ from the older one given the border edge f to add, S , and $MB = \text{MaxBorder}(S)$.

For efficient update, the algorithm uses a dynamic data structure \mathcal{D} for storing a set \mathcal{D} of candidate hyperedges, which is similar to the *DLX* (also known as “*Dancing Links*”) data structure by Knuth [83] as described in Sect. 5.1. For complexity analysis, recall that $\|N(S)\|$ denotes the sum $\sum_{e \in S} |N(e)|$ of neighbor hyperedges to S , where $\|S\| + \|N(S)\| = O(\|\mathcal{E}\|) = O(nm)$. Then, we have the next lemma.

Lemma 24. *Let $S \subseteq \mathcal{E}$ be a sub-hypergraph and $f \in R = (\mathcal{E} \setminus S)$ be a maximum border hyperedge of S . Given f , B , and R , the algorithm **UpdateMaxBorder** in Algorithm 7 computes the set $\text{MaxBorder}(S \cup \{f\})$ of all maximum border hyperedges of $S' = S \cup \{f\}$ in $O(\|S\| + \|N(S)\|)$ amortized time using $O(N)$ space and $O(N)$ preprocessing (at once in the initialization), where $B = \text{MaxBorder}(S)$ is the set of all maximum borders of S , and $N = \|\mathcal{E}(\mathcal{H})\|$.*

Proof. Consider Algorithm 7. During the computation of the recursive enumeration procedure, we maintain the pointers $\max(S)$, $2\max(S)$, and the dynamic data structure \mathcal{D} . From Lemma 23, Step 1 correctly updates the maximum and 2nd maximum leaves in S in constant time. When a new border f is added to S , the only borders to be changed are (i) f is removed, and (ii) all neighboring hyperedges of f , and (ii) all existing border edges that has a non-empty intersection to f other than its connection point to S . Step 2 handles these cases correctly. For time analysis of Step 2, we observe that during computation from the root hypothesis \emptyset to the current set S , any hyperedge

Algorithm 7: The algorithm for computing the border set of a sub-hypergraph

```

1 Procedure UpdateMaxBorder( $S, f$ : hyperedge,  $B, R \subseteq \mathcal{E}(\mathcal{H}), \mathcal{H}$ ), where  $\mathcal{D}$  is
   a dynamic data structure for storing a hyperedge subset  $\mathcal{D}$  in linear space
   supporting membership, insert, and delete in sublinear time  $t = f(m)$ .
   Output :  $MB(S') = \{f \in (\mathcal{E} \setminus S') \mid cnt(f, S') = 1, f = \max L(S')\}$ .
   Pre-conditions:  $S' = S \cup \{f\}$ ,  $f \in R$ ,  $B = MB(S)$ , and  $R = \mathcal{E}(\mathcal{H}) \setminus S$ .
   // Step 1: Update the maximum leaves.
2  $S' \leftarrow S \cup \{f\}$ ;
3 if  $f$  connects  $\ell_{\max} = \maxLeaf(S)$  then
   | // in  $O(1)$  time
4   | if  $f > 2\max(S)$  then
5   | |  $\max(S') \leftarrow f$  and  $2\max(S') \leftarrow \max(S)$ ;
6   | else
7   | |  $\max(S') \leftarrow \max(S)$  and  $2\max(S') \leftarrow 2\max(S)$ ;
8 else
   | // in  $O(1)$  time
9   | if  $f > \max(S)$  then
10  | |  $\max(S') \leftarrow f$  and  $2\max(S') \leftarrow \max(S)$ ;
11  | else
12  | |  $\max(S') \leftarrow \max(S)$  and  $2\max(S') \leftarrow 2\max(S)$ ;
   // Step 2: Update the maximum border set.
13  $MB(S') \leftarrow \emptyset$ ;
   // Step 2.1: Existing borders other than  $f$ .
14 foreach  $e$  in  $MB(S) \setminus \{f\}$  do
15   | if  $e > \max(S')$  then
16   | | Add  $e$  to  $MB(S')$ ; // Charge  $O(|MB(S)|)$  time to  $S$ .
   // Step 2.2: New borders connecting to  $f$ .
17 foreach vertex  $x \in f$  do // Charge  $O(f)$  time to  $S'$ 
18   | foreach hyperedge  $id\ e \in N(x, \mathcal{D})$  do // Charge  $O(1)$  time to  $e \in \mathcal{D}$ .
19   | |  $cnt[e] \leftarrow cnt[e] + 1$ ;
20   | | if  $cnt[e] = 1$  then
21   | | | Add  $e$  to the candidate set  $\mathcal{D}$ ; // Charge  $O(f(n))$  time to  $e$ .
22   | | | if  $e > \max(S')$  then
23   | | | | Add  $e$  to  $MB(S')$ ;
24   | | else if  $cnt[e] = 2$  then
25   | | | Remove  $e$  from candidate set  $\mathcal{D}$ ; // Charge  $O(f(n))$  time to  $e$ .
   // Note: each hyperedge is processed at most twice overall.

```

e in \mathcal{H} will be processed at most twice after initialization, that is, it is incremented to $\text{cnt}(e) = 1$ at the first time, and it is incremented to $\text{cnt}(e) = 2$ the second time. Then, it is removed from \mathcal{D} forever (otherwise a back tracking occurs). By using appropriate charging scheme of the cost to each edges in S , we can show that the amortized cost for Step 2 to obtain each S is at most $\|S\| + \|N(S)\| = O(nm) = O(N)$, where $\|S\| = \sum_{e \in S} |e|$. \square

From Lemma 24, we show the main theorem of this chapter.

Theorem 10 (The adaptive delay by the modified enumeration algorithm). *The algorithm **FastBergeEnum** of Fig. 8 finds all connected Berge-acyclic sub-hypergraphs contained in an input hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ without duplicates in $O(\|S\| + \|N(S)\|)$ delay (time per solution) using $O(N)$ space and $O(N)$ preprocessing, where $\|S\| = O(nm)$ is the total length of hyperedges in S , and $N = \|\mathcal{E}\|$ are the numbers of vertices and hyperedges, and the total size of the hyperedges in \mathcal{H} .*

Proof. The correctness of the algorithm **FastBergeEnum** is obvious from the correctness of the basic algorithm and the definition of *MaxBorder*. Moreover, the time complexity follows from Lemma 24. Hence, the statement holds. \square

From the theorem, we have the following corollary.

Corollary 11. *The class of all connected Berge-acyclic sub-hypergraphs contained in an input hypergraph \mathcal{H} can be enumerated in delay that depends only on the size $\|S\|$ of a discovered subset S using polynomial space and preprocessing in the input size $\|\mathcal{E}(\mathcal{H})\|$.*

Algorithm 8: The modified algorithm `FastBergeEnum` for enumerating all connected, Berge-acyclic sub-hypergraphs based on the reverse search

```

1 Procedure FastBergeEnum( $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ )
2   foreach hyperedge  $e \in \mathcal{E}(\mathcal{H})$  do
3      $MB_f \leftarrow \{f \in \mathcal{E}(\mathcal{H}) \mid (|f \cap e| = 1)\};$ 
4      $R_f \leftarrow \mathcal{E}(\mathcal{H}) \setminus \{f\};$ 
5     FastRec( $\{e\}, MB_f, R_f, \mathcal{H}$ );
6 Subprocedure FastRec( $S, MB, R \subseteq \mathcal{E}(\mathcal{H}), \mathcal{H}$ )
7   Invariants      :  $MB = \text{MaxBorder}(S)$  and  $R = \mathcal{E}(\mathcal{H}) \setminus S$  hold.
8   Output  $S$ ;
9   foreach border hyperedge  $f \in MB$  do           // Generation of children
10     $S' \leftarrow S \cup \{f\};$ 
11     $R' \leftarrow R \setminus \{f\};$ 
12    Incrementally compute  $MB' = \text{MaxBorder}(S', \mathcal{H})$  from  $f$ ,  $MB$ , and  $R$ ;
13    FastRec( $S', MB', R', \mathcal{H}$ );

```

Chapter 6

K -subtree Bit Enumeration

In this chapter, we study the efficient computation of the frequency-based tree similarities using classes of *ordered trees*. They are useful for modeling semi-structured documents such as HTML and XML, chemical compounds, natural language data, and Web access logs. In one direction, ordered tree similarities based on substructures have been extensively studied [81, 85, 86], where the focuses are on substructures of restricted forms such as paths [81] and q -grams [85]. In other direction, Kashima and Koyanagi [77] presented an efficient dynamic programming algorithm to compute the *ordered subtree kernel* of two ordered trees T_1 and T_2 in $O(|T_1| \times |T_2|)$ time using general ordered subtrees of *unbounded size*. Besides the efficient DP algorithms for ordered trees of unbounded size [77], some authors pointed out the usefulness of the semi-structured features using bounded sized substructures [87]. However, it does not seem easy to extend the DP algorithm [77] for *bounded sized* ordered trees.

The *enumeration-based approach* [87, 6, 133] is another way of computing such a tree distance based on a general class of substructures, which is a simple and flexible approach that one uses a *pattern enumeration algorithm* [6, 152, 161], to find all substructures contained in an input data to construct a feature vector, and then to solve a variety of tasks for information retrieval, data mining, and machine learning using

similarity measure obtained from the constructed feature vectors. One problem in this approach is the high computational complexity of enumerating all small substructures. Hence, our goal is to devise efficient algorithms for frequency-based similarity by employing the recent development of efficient enumeration and mining algorithms for semi-structured data [6, 152, 161].

In this chapter, we study efficient computation of tree similarity between two ordered trees using as features the class of *bounded sized ordered subtrees in unrestricted shape*. We present two new efficient algorithms for enumerating the *compressed bit-signatures* of all ordered k -subtrees in an input ordered tree using bit-parallel speed-up technique. The first one runs in $O(k)$ time per signature, and the second one runs in constant time per signature using $O(n)$ time preprocessing [5, 71]. From these compressed signatures, we can quickly compute the tree similarity between two ordered trees. We note that these algorithm are the first *compressed pattern enumeration algorithms* [155] for a subclass of trees and graphs. They directly enumerate the compressed representation of all substructures by incrementally constructing their compressed form on-the-fly without encoding/decoding.

Finally, we ran the experiments on real and artificial datasets to evaluate the proposed methods. We observed that the improved versions of the algorithms equipped with our bit-parallel speed-up technique showed around 36% speed-up from the algorithm without speed-up.

6.1 Preliminaries

6.1.1 Bit signatures

In what follows, vertices in a tree are numbered in the DFS manner. As the succinct representation of ordered trees, the *balanced parentheses representation* (BP) [5] of an ordered tree T of k vertices is a bit sequence $BP(T) = b_{2k-1} \cdots b_0$ defined by the depth-

first traversal of T starting from $\text{root}(T)$ the left and right parentheses, “(” = 0 and “)” = 1, when it visits a vertex at the first and last times, respectively. We call $BP(T)$ the bit signature of T . For example, in Fig. 6.1, the BP of a tree $S_4 = \{2, 3, 8, 9, 10\}$ of size 5 is $BP(S_4) = ((()((())))$.

For each vertex $v \in T$, $lpos(v)$ and $rpos(v) \in [0, 2k - 1]$ denotes the bit positions for the left and right parentheses in $BP(S)$ corresponding to v , respectively. For any subset $R \subseteq S$, $RPOS(R)$ denotes the bit-vector $X \in \{0, 1\}^{2k}$ such that, for every $v \in S$, $X[rpos(v)] = 1$ iff $v \in R$. Now, we state our problem.

Problem 3 (compressed subtree enumeration). *Given an input tree T and an integer k , enumerate all k -subtrees in T in the form of bit signature without duplicates.*

Model of Computation

We assume the Word RAM [5, 71] with standard bit-wise Boolean and arithmetic operations (“+” and “*”) on $w = \Theta(\log n)$ bits registers including *bitwise* AND “&”, *bitwise* OR “|”, *bitwise* NOT “~”, *left shift* “<<”, and *right shift* “>>”, where n is an input size. We write a constant variable-length bit-vector as “1011”. In this chapter, a bit vector of length L is written as $X = b_{L-1} \cdots b_0 \in \{0, 1\}^L$, where the *most significant bit* (MSB) b_{L-1} and the *least significant bit* (LSB) b_0 come in this order. For every i , we define the length and the i -th bit of B by $|B| = L$ and $B[i] = b_i$, respectively.

6.1.2 Tree similarity

In this subsection, we give the tree similarity for ordered trees [77, 86]. Let T be an input ordered tree and $\mathcal{S} = \{S_1, \dots, S_M\} \subseteq \mathcal{T}$, $M \geq 1$, be a class of possible subtrees in T . Each elements of \mathcal{S} are called a *subtree-feature*. The *subtree-feature vector* of T based on \mathcal{S} is the vector $\phi^{\mathcal{S}}(T)$ of the number of counts that subtrees of \mathcal{S} appear in T . Below, we will omit the superscript \mathcal{S} if it is clear from context. Formally, the

subtree-feature vector $\phi(T)$ for T based on \mathcal{S} is defined by

$$\phi(T) = (f_1(T), \dots, f_M(T)) \in \mathbb{N}^M,$$

where \mathbb{N} is the set of natural numbers including zero and for every $i \in [1, M]$, $f_i(T) = \text{Occ}(S_i, T)$ is the number of all occurrences of the i -th subtree S_i in T . Then, we consider the tree similarity $\text{Sim}(T, T')$ between T and T' in one of the following forms [96]:

- L_p -tree distance for every $p = 1, 2, \dots$:

$$\text{Sim}(T, T') = \|\phi(T) - \phi(T')\|_p = \left(\sum_i |f_i(T) - f_i(T')|^p \right)^{1/p}.$$

- Cosine-tree distance:

$$\text{Sim}(T, T') = \text{Cosine}(\phi(T), \phi(T')) = \frac{\sum_i f_i(T) \times f_i(T')}{(\sum_i f_i(T))^{\frac{1}{2}} \times (\sum_i f_i(T'))^{\frac{1}{2}}}$$

Once the feature vectors $\phi(T)$ and $\phi(T')$ are computed by a subtree enumeration algorithm for class \mathcal{S} , $\text{Sim}(T, T')$ can be computed in linear time in the length of the vectors. In Fig.6.1, we show examples of an ordered tree T_1 , the corresponding k -subtrees, and the feature vector $\phi^{\mathcal{S}_k}(T_1)$ for T_1 based on all k -subtrees for $k = 5$.

6.2 Enumeration of At-Most K -subtrees in a Tree

In Sect. 6.2 and Sect. 6.3, we present algorithms for computing the subtree-feature vector of T based on efficient compressed subtree enumeration.

The first algorithm that we present in this section is the compressed enumeration version of the constant delay enumeration algorithm for uncompressed subtrees of at

most size k in an ordered tree by [6, 106, 161].

6.2.1 The outline of the enumeration algorithm

In Algorithm 9, we show our algorithm **EnumAtMost** for at most k -subtrees and its subprocedure **RecAtMost**. This is a simple backtrack algorithm, which starts from a singleton tree as 0-subtree, and recursively expands the current $(i - 1)$ -subtree by attaching a new vertex u to some vertex $v = pa(u)$ on the current rightmost branch $RMB(S)$ to generate a new i -subtrees, until its size i becomes k (See [6]). Then, we say that some vertex $v \in T \setminus S$ can be added to S as a child of a vertex $u = pa(v)$ on $RMB(S)$ if v is the younger than any child of u contained in S . Such a parent vertex u on $RMB(S)$ is called the *extension point* and v is called the associated new child. If there is no such a vertex u , then the algorithm backtracks to the parent subtree. The *extension point set* is the set $XP(S) \subseteq RMB(S)$ of all extension points of S . We give the definition of $XP(S)$ as follows.

Definition 11. For any $u \in RMB(S)$, $u \in XP(S)$ if there exists some $v \in T \setminus S$ such that (i) $v > \max(S)$ and (ii) v is younger than the youngest child of u in S .

Example 9. For the 5-subtree $S_3 = \{2, 3, 8, 9, 10\}$ in Fig.6.1, $Lv(S_4) = \{3, 10\}$ and $Bd(S_4) = \{4, 5, 11, 12, 13\}$, $RMB(S_4) = \{2, 8, 9, 10\}$, and $XP(S_4) = \{8, 9\}$.

We will show how to incrementally maintain the extension set $XP(S)$ by growing S . For a singleton tree S consisting with the root $r = root(S)$ only, if r has a child in T then $XP(S) = \{r\}$, and otherwise $XP(S) = \emptyset$. For a subtree with more than one vertices, we have the next lemma.

Lemma 25. Let S be any k -subtree of T with $k \geq 2$. Suppose that $k \geq 2$ and a k -subtree $R = S \cup \{v\}$ is obtained from a $(k - 1)$ -subtree S by attaching a new child v to its extension point $u = pa(v) \in XP(S)$. Then, $XP(R)$ is the set of vertices that satisfies the following (a)–(c):

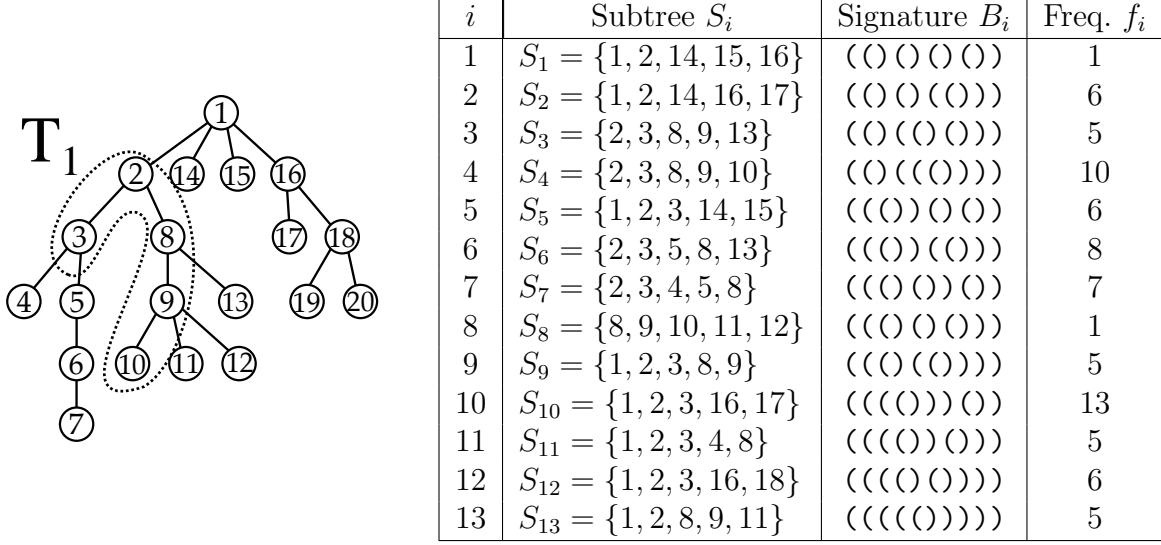


Figure 6.1: An ordered tree T_1 , the corresponding k -subtrees, and the feature vector $\phi^{S_k}(T)$ for T_1 , where $k = 5$. The set of vertices surrounded by a dashed line indicates the subtree $S_4 = \{2, 3, 8, 9, 10\}$, which has occurrences $S_4^1 = \{1, 15, 16, 18, 19\}$ and $S_4^2 = \{2, 3, 8, 9, 12\}$ in T isomorphic to itself.

Algorithm 9: The algorithm **EnumAtMost** for computing the feature vector H for the bit signatures of all subtrees with at most k vertices in an input tree T

```

1 Procedure EnumAtMost( $T, k$ )
2    $H \leftarrow \emptyset$ ; // A hash table  $H$  representing a feature vector
3   for  $r \leftarrow 1, \dots, n$  do
4     Initialize bit-vectors  $B$  and  $X$ ;
5     RecAtMost( $\{r\}, T, k$ );
6   return  $H$ ;

7 Procedure RecAtMost( $S, T, k$ )
8   if  $H[S]$  is defined then
9      $H[S] \leftarrow H[S] + 1$  else  $H[S] \leftarrow 1$ ;
10  Output BS( $S$ );
11  if  $|S| = k$  then
12    return;
13  foreach extension point  $v$  on  $RMB(S)$  do
14    Attach a new leaf  $u$  to  $v$  as the youngest child;
15    Let  $S \cup \{u\}$  be the resulting subtree;
16    RecAtMost( $S \cup \{u\}, T, k$ );

```

- (a) For the parent, $pa(v) \in XP(R)$ iff v has a properly younger sibling in T .
- (b) For the child, $v \in XP(R)$ iff v has some child in T .
- (c) For any old extension point $x \in XP(S)$ other than $pa(v)$, $x \in XP(R)$ iff x is an ancestor of $pa(v)$.

Proof. We can immediately follow this lemma from the DFS-numbering. \square

In condition (c) of the above lemma, we note that any extension point x is either an ancestor or a descendant of $pa(v)$ in $XP(S)$ since $XP(S)$ is a subset of $RMB(S)$, a branch in S .

6.2.2 Fast update of bit signatures

In our bit-parallel implementation of `EnumAtMost`, for each k -subtree, we maintain two bit-vectors of length $2k$, $B = BP(S)$ and $X = RPOS(XP(S))$, that represent the current subtree S and its extension point set $XP(S)$, respectively. For simplicity, we first describe the algorithm with bit-vectors whose length is no larger than the word length $w = \Theta(\log n)$. We efficiently update the bit-vectors B and X as follows.

Let $i \geq j$ and $ONE_{i:j}^\ell = 0^{i-1}1^{j-1+1}0^{\ell-j} \in \{0,1\}^\ell$ be the bit-mask of length ℓ whose i to j bits are filled with 1 bits and the other bits are filled with 0 bits, which can be computed by shift and subtraction in constant time for $i, j = O(\log n)$.

First, we initialize the bit-vectors B and X for the sets $S = \{r\}$ and $XP(S)$ by the following code: $B \leftarrow \text{"01"}; \text{ if } r \text{ has a child on } T \text{ then } X \leftarrow \text{"01"} \text{ else } X \leftarrow \text{"00"};$

Next, the following code correctly updates the bit-vectors B and X when we compute the extension point set $XP(S \cup \{v\})$ for the new subtree $S \cup \{v\}$ from $XP(S)$ for the old one S , where $q = rpos(v)$ and $\ell = len(B)$:

- B is updated as follows.

$$1 \ B \leftarrow (B \ \& \ ONE_{\ell-1:q+1}^\ell) \ll 2 \quad | \quad (\text{"01"} \ll q) \quad | \quad (B \ \& \ ONE_{q:0}^\ell);$$

- X is updated as follows.

```

1  $X \leftarrow \{ ("1" \ll q - 1) \text{ if } v \text{ has a properly younger sibling } \} \quad // \text{ a parent}$ 
   |  $\{ ("1" \ll q) \text{ if } v \text{ has some child } \} \quad // \text{ a child}$ 
   |  $(X \& ONE_{q-1:0}^\ell) \gg 2 ; \quad // \text{ others}$ 

```

From Lemma 25, we have the following lemma.

Lemma 26 (Update in the small subtree case). *If $2k \leq w$, the above codes correctly updates the bit-vectors B and X in constant time using $O(1)$ words.*

Proof. This lemma is trivial since each bit operation in the above codes can be done in $O(1)$ time. \square

Lemma 27 (Update in the large subtree case). *If $k = O(2^w)$, the bit-vectors B and X can be correctly updated in constant time using $O(k/w)$ words.*

Proof. We represent bit-vectors B and X as a doubly linked list of $b = O(\log n)$ -bits blocks, each of which are maintained to store consecutive bits of length $\lceil b/2 \rceil < \ell \leq b$ (bits) similarly to [71]. In the update of B and X , we need to update only constant number of blocks, and thus, takes constant time. \square

From Lemma 26 and Lemma 27, we can get the following theorem.

Theorem 12 (Compressed enumeration of at most k -subtrees). *For every $k \geq 1$, Algorithm 9 enumerates all compressed representations of the at most k -subtrees appearing in an ordered tree T in $O(1)$ time per solution, generated on the bit-vector $B \in \{0, 1\}^{2k}$, using $O(k/w)$ words of space in addition to the space for an enumeration algorithm.*

From the above theorem, we observe that for every $k \geq 1$, all compressed representations of exact k -subtrees in T can be enumerated in $O(k)$ time per compressed representation using the same amount of space as above.

6.3 Enumeration of Exact K -subtrees in a Tree

The second algorithm that we present in this section is the compressed enumeration version of the constant delay enumeration algorithm for uncompressed k -subtrees in an ordered tree introduced in Chap. 3.

6.3.1 The outline of the algorithm

The basic idea of our algorithm is as follows: Given a k -subtree S in an input tree T , we can obtain the other k -subtree S' from S by deleting one vertex from S and adding one vertex to S .

By repeating this process recursively using backtracking, for each vertex r in T , starting from the lexicographically least k -subtree with r as its root, we can enumerate all k -subtrees with root r appearing in T by recursively transforming the current k -subtree by the above process. This algorithm runs in $O(1)$ time per k -subtree by maintaining the vertex lists $DL(S) \subseteq Lv(S)$ and $AL(S) \subseteq Bd(S)$ to delete and to add, respectively. In Algorithm 10, we show our algorithm **EnumExact** and its subprocedure **RecExact**.

6.3.2 Fast update of bit signatures

In the implementation with bit-operations, we use three bit-vectors B , L , and $A \in \{0, 1\}^*$, where B is the *BP-vector* as defined in the previous section, L is the *leaf-vector* representing the set of leaves to delete, and A is the *add-vector* representing the set of vertices to add.

In this section, we give the efficient method for updating bit-vectors using bit parallel technique. A vertex v is an *exact extension point* in S if one of children of v can be attached to S . We define the sets $AL(S)$ and $DL(S)$ of vertices to add and to delete,

Algorithm 10: The algorithm **EnumExact** for computing the feature vector H for the bit signatures of all subtrees with exactly k vertices in an input tree T based on constant delay enumeration

```

1 Procedure EnumExact( $T, k$ )
2    $H \leftarrow \emptyset$ ; // A hash table  $H$  representing a feature vector
3   Number the vertices of  $T$  by the DFS-numbering;
4   Compute the initial  $k$ -subtree  $\mathcal{I}_k$ ;
5   Initialize the related lists and pointers;
6   RecExact( $\mathcal{I}_k, B, L, X; T, k$ );
7   return  $H$ ;

8 Procedure RecExact( $S, B, L, X; T, k$ )
9   Output BS( $S$ );  $p \leftarrow \text{MSB}(L)$ ;
10  for each  $\ell \in \text{Dellist}(S)$  do
11    foreach  $\beta \in \text{AddList}(S)$  such that  $\beta \notin \text{Ch}(\max(Lv(S)))$  do
12       $S \leftarrow \text{Child}_1(S, \ell, \beta)$  by updating the related lists and pointers;
13      Update bit sequences  $B, L, X$ ;
14      RecExact( $S, B, L, X; T, k$ );
15       $S \leftarrow \mathcal{P}^1(S)$  by restoring the related lists and pointers;
16      Restore bit sequences  $B, L, X$ ;
17      Modify  $X$ ;
18    Proceeds  $p$ ; // to the next leaf position in  $L$ 

19  if  $S$  is a  $k$ -pre-serial tree then
20     $S \leftarrow \text{Child}_2(S)$  by updating the related lists and pointers;
21    Update bit sequences  $\text{sig}(S), A, L$ ;
22    RecExact( $S, B, L, X; T, k$ );
23     $S \leftarrow \mathcal{P}^2(S)$  by restoring the related lists and pointers;
24    Restore bit sequences  $\text{sig}(S), A, L$ ;

```

and the set $EXP(S)$ of exact extension points by

$$\begin{aligned} DL(S) &= \{x \in Lv(S) \mid x < \text{minbord}(S)\}. \\ AL(S) &= \{x \in Bd(S) \mid x > \text{maxleaf}(S)\}. \\ EXP(S) &= \{x \in T \mid x = \text{pa}(v), v \in AL(S)\}. \end{aligned}$$

We give the following recurrence relation for $Lv(S)$, $Bd(S)$, and $EXP(S)$. In this subsection, \leq denotes the DFS-ordering on T .

Lemma 28. *Then, set is defined for any subset S .*

- (a) *If $S = \mathcal{I}_k$ is an initial k -subtree rooted at u , then $AL(\mathcal{I}_k)$ is the set $XP(\mathcal{I}_k)$ of all extension points, and $DL(\mathcal{I}_k)$ is the set $Lv(\mathcal{I}_k)$ of all leaves.*
- (b) *Let S be any k -subtree, $v \in AL(S)$, and $u \in DL(S)$ such that v is not a child of u . Suppose that $S' = (S \setminus \{u\}) \cup \{v\}$. Then, the following conditions hold:*
 - (i) $Lv(S') = \{x \in Lv(S) \mid x \neq u\} \cup \{x \in T \setminus Lv(S) \mid Ch(x) \cap S = \{u\}\}.$
 - (ii) $DL(S') = \{x \in DL(S) \mid x < u\} \cup \{x \in T \setminus DL(S) \mid Ch(x) \cap S = \{u\}\}.$
 - (iii) $AL(S') = \{x \in AL(S) \mid x > v\} \cup \{x \in T \setminus AL(S) \mid x \in Ch(v)\}.$

Proof. (a) This is obvious from the DFS-ordering. (b) (i) is obvious. (ii) The smaller vertices than u in $DL(S)$ remain in $DL(S')$. On the other hand, the larger vertices do not belong to $DL(S')$ since u is the new smallest border in S' . Moreover, if x has the only child u in S then x is in $DL(S')$ since x is a leaf in S' . (iii) The larger vertices than v in $AL(S)$ remain in $AL(S')$. On the other hand, the smaller vertices do not belong to $AL(S')$ since v is the new largest leaf in S' . Moreover, if children of v are larger than v and are in $Bd(S')$. \square

During the enumeration, the algorithm explicitly maintains the lists $Lv(S)$, $Bd(S)$, and $EXP(S)$. Using these lists and the pointers to the maximum leaf $\text{maxleaf}(S)$

and to the minimum border vertex $\text{minbord}(S)$, the algorithm implicitly represents the lists $DL(S)$ and $AL(S)$.

Next, we consider the generation of children of type I from Line 10 to Line 18 in Algorithm 10, and give the bit-parallel implementation of the update procedure for bit-vectors B , L , X , and pointers p and q to them, while the lists $Lv(S)$, $Bd(S)$ and $EXP(S)$ are maintained by the algorithm. During the enumeration, we maintain B , L , and X such that $B = BP(S)$, $L[rpos(v)] = 1$ iff $v \in Lv(S)$, and $X[rpos(v)] = 1$ iff $v \in EXP(S)$ for every $v \in T$. For initialization, we set $B = BP(\mathcal{I}_k)$, $L = RPOS(Lv(\mathcal{I}_k))$, and $X = RPOS(EXP(\mathcal{I}_k))$ in $O(k)$ time by traversing the initial k -subtree \mathcal{I}_k .

Definition 12 (Update for children of type I). *Suppose that we generate a child k -subtree $S' = (S \setminus \{u\}) \cup \{v\}$ of type I from the parent S . Then, we update the bit-vectors B , L and X as follows, where $p = rpos(u)$ and $\ell = \text{len}(B)$:*

- *The right position $q = rpos(v)$ can be computed from the bit-vector X using MSB by the following code:*
 - 1 $q \leftarrow \text{MSB}(X);$
- *B is updated by deleting the two bits “01” from right position p for vertex u , and inserting the two bits “01” for vertex v at right position $q - 1$.*
 - 1 $B \leftarrow (B \& \text{ONE}_{\ell-1:p+2}^\ell) \mid (B \& \text{ONE}_{p-1:q+1}^\ell) \gg 2;$
 - 2 $B \leftarrow B \mid (\text{“01”} \ll q) \mid (B \& \text{ONE}_{q:0}^\ell);$
- *L is updated similarly to B . In addition, the two bits surrounding the delete position for u are overwritten with “01”:*
 - 1 $L \leftarrow (L \& \text{ONE}_{\ell-1:p+3}^\ell) \mid (L \& \text{ONE}_{p-1:q+1}^\ell) \ll 2 \mid (\text{“01”} \ll q)$
 $\mid (L \& \text{ONE}_{q-1:0}^\ell) \mid \{ (\text{“01”} \ll p + 1) \text{ if } Ch(pa(u)) = \{u\} \};$
- *X is updated similarly to B . In addition, the two bits surrounding the delete position for u are overwritten with “01”:*

1 $X \leftarrow (X \& ONE_{q-1:0}^\ell) \mid \{ ("1" \ll q) \text{ if } v \text{ has a child } \}$
 $\mid ("1" \ll q-1) \text{ if } (\exists \text{ younger sibling } r \text{ of } v) r \notin S;$

Moreover, after the for-loop at line 17, we update X by deleting the extension point at the highest position one by one:

1 $X \leftarrow X \& (\sim ("1" \ll (\text{MSB}(X) - 1))) \text{ if } v \text{ has no younger sibling in } T;$

- We proceed the pointer $p = \text{rpos}(u)$ at line 18 by:

1 $p \leftarrow \text{MSB}(L \& ONE_{1:p-1}^\ell);$

Next, the code from Line 19 to Line 24 generates the children of type II updating the bit-vectors B , L , and X .

Definition 13 (Update for children of type II). *Suppose that we generate a child k -subtree $S' = (S \setminus \{u\}) \cup \{v\}$ of type II from the parent S . Then, we update the bit-vectors B , L and X as follows:*

- The right position q can be computed by

1 $q \leftarrow \text{MSB}(X);$

- B is updated by deleting the most left bit of B and the most right bit of B , and inserting the two bits “01” for vertex v position q .

1 $B \leftarrow (B \& ONE_{2k-2:q+1}^{2k}) \ll 1 \mid ("01" \ll q-1) \mid (B \& ONE_{q:1}^{2k}) \gg 1;$

- L is updated similarly to B . In addition, the right position bit of the inserting vertex v , is overwritten with “0”.

1 $L \leftarrow (L \& ONE_{2k-2:q+1}^{2k}) \ll 1 \mid ("01" \ll q-1) \mid (L \& ONE_{q-1:1}^{2k}) \gg 1;$

- X is updated by overwriting bits in the left of q with “0”. In addition, two bits are overwritten with “1” if corresponding vertices satisfy some conditions.

1 $X \leftarrow (X \& ONE_{q-1:1}^{2k}) \gg 1 \mid \{ ("1" \ll q-1) \text{ if } v \text{ has a child; } \}$
 $\mid ("1" \ll q-2) \text{ if } (\exists \text{ younger sibling } r \text{ of } v) r \notin S;$

In the small tree case that $2k \leq w$, it follows from Lemma 28 that the above procedure correctly updates the data structure in constant time per iteration using $O(1)$ words. In the large tree case that $k = O(2^w)$, a similar discussion to Lemma 27 shows that the procedure also run in constant time using $O(k/w)$ words. Therefore, we have the following theorem.

Theorem 13 (Compressed enumeration of exact k -subtrees). *Let T be an input tree and k be a positive integer. The algorithm `EnumExact` in Algorithm 10 enumerates all compressed representations of the exact k -subtrees appearing in T in $O(1)$ time per compressed representation, generated on the bit-vector $B \in \{0, 1\}^{2k}$, using $O(k/w)$ words of space in addition to the space for an enumeration algorithm.*

From the above theorem, we obtained a constant-delay algorithm for compressed enumeration for exact k -subtrees, which improves on the $O(k)$ -delay algorithm in Sect. 6.2 by a factor of $O(k)$.

6.4 Experiments

In the experiments, we compared the running time of the algorithms in Sect. 6.2 and Sect. 6.3 on artificial and real datasets. We implemented in C++ the algorithms `EnumAtMost` in Sect. 6.2 and `EnumExact` in Sect. 6.3, denoted by `Atmost(α)` and `Exact(α)`, respectively, where α indicates the types of algorithms as follows:

- “Enum” enumerates subtrees without printing them.
- “Naive” is the original algorithm that first enumerates a subtree and then computes its bit signature.
- “Fast” is the modified algorithm that directly enumerates the bit signature of a subtree with bit-parallel signature maintenance.

The algorithms were compiled by g++ 4.2.1 and were run on a PC (CPU Intel[®] Xeon(R) 3.6GHz, 34GB RAM) operating on Ubuntu OS 13.04.

6.4.1 Comparison of algorithms on real data

As input, we use a phylogenetic tree of influenza virus of $n = 4240$ vertices, which was constructed from virus data in NCBI Influenza Virus Resource¹ by neighbor-joining method. In Fig.6.2, we show the running time of algorithms for computing the feature vector $\phi(T)$ of an input tree T varying the size of subtrees for $k = 15$ to 19. From this figure, for each of **Exact** and **Atmost**, the fast version (**Fast**) was faster than the naive version (**Naive**). For example, the speedup ratio for $k = 19$ were 36% for **Exact**, and 22% for **Atmost**. It depends on the type of update α which is faster between **Exact** and **Atmost**. In the case of **Exact**, the overhead of computing bit signatures over enumeration only (**Enum**) are 6 times for **Fast** and 9.5 times for **Naive**.

6.4.2 Comparison of algorithms on artificial data

We used a artificial tree with size $n = 35$, which has depth one and consists of a root vertex and 34 leaves. Fig.6.3 shows the result of experiment. The fastest algorithm was **Exact(Fast)**, which was 34% faster than **Exact(Naive)** for $k = 18$.

6.4.3 Computing the gram matrix of a set of trees

To evaluate the usefulness of our algorithms in the context of tree mining [77, 87], we applied **Exact(Fast)** to similarity matrix computation [85, 86]. We computed $M = 70,532$ dependency trees, one tree per one Japanese sentence, in total size 1,140,098 vertices and 3.8 MB from a Japanese newspaper corpus² in 44.9 MB by CaboCha.³

¹<http://www.ncbi.nlm.nih.gov/genomes/FLU/>

²http://www.ndk.co.jp/yomiuri/e_yomiuri/e_index.html

³<http://code.google.com/p/cabocha/>

Their average and standard deviation sizes are 16.16 and 12.65 (vertices). Applying **Exact(Fast)** to this dataset with $k = 4$, we computed the $M \times M$ -similarly matrix for 4-subtrees using cosine-tree distance in 1,276.12 seconds, where only 0.1% (1.61 seconds) of the time was spent for computing feature vectors and 99.9% for matrix computation.

6.4.4 Summary of experimental results

Overall, the proposed method (**Fast**) achieved around 22% to 30% speedup over the naive method (**Naive**). From the last experiment, the proposed method seems to have reasonable performance for data mining from middle size datasets.

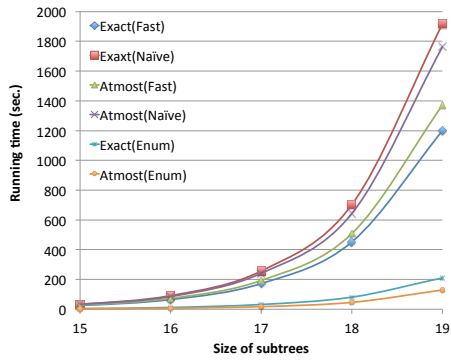


Figure 6.2: The running time against the subtree size k on the real phylogenetic tree with $n = 4240$ vertices.

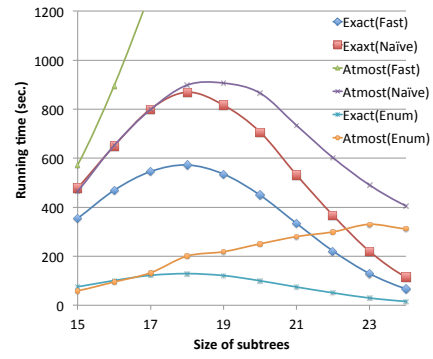


Figure 6.3: The running time against the subtree size k on the artificial tree with $n = 35$ vertices and depth one.

Chapter 7

Conclusion and Future Work

In this thesis, we studied the enumeration problems for acyclic substructures. By combining the basic techniques and non-trivial amortized analysis, we developed efficient enumeration algorithms for our problems. Finally, we show further directions of this research area as follows:

- (I) The best known k -subtree enumeration problem runs in $O(k)$ time per solution [46]. Can we enumerate k -subtrees in general graphs in constant delay?
- (II) Does there exist an output-sensitive enumeration algorithm for maximal induced trees in a general graph? In addition, a maximal subtree corresponds to a spanning tree. What is the essential difference between enumeration problems for induced graphs and for subgraphs?
- (III) In Chap. 5, we proposed an enumeration algorithm for Berge-acyclic sub-hypergraph in polynomial amortized time per solution. In addition, Daigo and Hirata [31] proposed an enumeration algorithm for maximal α -acyclic sub-hypergraphs. Can we also enumerate Berge-, β -, and γ -acyclic sub-hypergraphs in polynomial amortized time per solution?

- (IV) Does there exist an universal complexity analysis technique for enumeration algorithms for connected and acyclic substructures?
- (V) Can we develop efficient exact exponential algorithms [48] using enumeration and amortization analysis technique?

Bibliography

- [1] E. A. Akkoyunlu. “The Enumeration of Maximal Cliques of Large Graphs”. *SIAM Journal on Computing*, 2(1):1–6. 1973. DOI: 10.1137/0202001.
- [2] Hiroki Arimura and Takeaki Uno. “An Efficient Polynomial Space and Polynomial Delay Algorithm for Enumeration of Maximal Motifs in a Sequence”. *Journal of Combinatorial Optimization*, 13(3):243–262. 2006. DOI: 10.1007/s10878-006-9029-1.
- [3] Hiroki Arimura and Takeaki Uno. “Mining Maximal Flexible Patterns in a Sequence”. In *Proceedings of JSAI 2007 Conference and Workshops on New Frontiers in Artificial Intelligence*. Lecture Notes in Computer Science. Vol. 4914, pages307–317. Springer Berlin Heidelberg, 2007. DOI: 10.1007/978-3-540-78197-4_29.
- [4] Hiroki Arimura and Takeaki Uno. “Polynomial-Delay and Polynomial-Space Algorithms for Mining Closed Sequences, Graphs, and Pictures in Accessible Set Systems”. In *Proceedings of the SIAM International Conference on Data Mining, SDM 2009*, pages1088–1099. 2009. DOI: 10.1137/1.9781611972795.93.
- [5] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiro Sadakane. “Succinct Trees in Practice”. In *Proceedings of the Twelfth Workshop on Al-*

- gorithm Engineering and Experiments, ALENEX 2010*, pages84–97. 2010. DOI: 10.1137/1.9781611972900.9.
- [6] Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroki Arimura, Hiroshi Sakamoto, and Setsuo Arikawa. “Efficient Substructure Discovery from Large Semi-structured Data”. In *Proceedings of the Second SIAM International Conference on Data Mining, SDM 2002*, pages158–174. 2002. DOI: 10.1137/1.9781611972726.10.
 - [7] David Avis and Komei Fukuda. “Reverse Search for Enumeration”. *Discrete Applied Mathematics*, 65(1-3):21–46. 1996. DOI: 10.1016/0166-218X(95)00026-N.
 - [8] Manu Basavaraju, Pinar Heggernes, Pim van ’t Hof, Reza Saei, and Yngve Villanger. “Maximal Induced Matchings in Triangle-Free Graphs”. In *Proceedings of the 40th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2014*. Lecture Notes in Computer Science. Vol. 8747, pages93–104. Springer International Publishing, 2014. DOI: 10.1007/978-3-319-12340-0_8.
 - [9] Paul Beame and Faith E. Fich. “Optimal Bounds for the Predecessor Problem and Related Problems”. *Journal of Computer and System Sciences*, 65(1):38–72. 2002. DOI: 10.1006/jcss.2002.1822.
 - [10] Richard Beigel. “Finding Maximum Independent Sets in Sparse and General Graphs”. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1999*, pages856–857. 1999.
 - [11] Claude Berge and Edward Minieka. *Graphs and Hypergraphs*. North-Holland, 1973.
 - [12] Philip Bille. “A Survey on Tree Edit Distance and Related Problems”. *Theoretical Computer Science*, 337(1-3):217–239. 2005. DOI: 10.1016/j.tcs.2004.12.030.

- [13] Etienne Birmelé, Rui A. Ferreira, Roberto Grossi, Andrea Marino, Nadia Pisanti, Romeo Rizzi, and Gustavo Sacomoto. “Optimal Listing of Cycles and st-Paths in Undirected Graphs”. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013*, pages 1884–1896. 2013. DOI: 10.1137/1.9781611973105.134.
- [14] Endre Boros, Khaled M. Elbassioni, and Vladimir Gurvich. “Transversal hypergraphs to perfect matchings in bipartite graphs: Characterization and generation algorithms”. *Journal of Graph Theory*, 53(3):209–232. 2006. DOI: 10.1002/jgt.20180.
- [15] Endre Boros, Khaled M. Elbassioni, Vladimir Gurvich, and Leonid Khachiyan. “An Efficient Incremental Algorithm for Generating All Maximal Independent Sets in Hypergraphs of Bounded Dimension”. *Parallel Processing Letters*, 10(4):253–266. 2000. DOI: 10.1142/S0129626400000251.
- [16] Endre Boros, Khaled M. Elbassioni, Vladimir Gurvich, and Leonid Khachiyan. “Generating Maximal Independent Sets for Hypergraphs with Bounded Edge-Intersections”. In *Proceedings of the Sixth Latin American Symposium on Theoretical Informatics, LATIN 2004*. Lecture Notes in Computer Science. Vol. 2976, pages 488–498. Springer Berlin Heidelberg, 2004. DOI: 10.1007/978-3-540-24698-5_52.
- [17] Coen Bron and Joep Kerbosch. “Algorithm 457: Finding All Cliques of an Undirected Graph”. *Communications of the ACM*, 16(9):575–577. Sept. 1973. DOI: 10.1145/362342.362367.
- [18] Samuel Rota Bulò, Andrea Torsello, and Marcello Pelillo. “A Continuous-Based Approach for Partial Clique Enumeration”. In *Proceedings of the Sixth IAPR-TC-15 International Workshop on Graph-Based Representations in Pattern Recog-*

- nition, GbRPR 2007*. Lecture Notes in Computer Science. Vol. 4538, pages61–70. Springer Berlin Heidelberg, 2007. DOI: 10.1007/978-3-540-72903-7_6.
- [19] R. N. Burns and C. E. Haff. “A Ranking Problem in Graphs”. In *Proceedings of the Fifth Southeast Conference on Combinatorics, Graph Theory and Computing*. Congressus Numerantium 19, pages461–470. Utilitas Mathematics Publication, 1974.
 - [20] P. M. Camerini, L. Fratta, and F. Maffioli. *The K Shortest Spanning Trees of a Graph*. Tech. rep. Int.Rep. 73-10, IEEE-LCE Politecnico di Milano, Italy, 1974.
 - [21] Kevin Cattell, Frank Ruskey, Joe Sawada, Micaela Serra, and C. Robert Miers. “Fast Algorithms to Generate Necklaces, Unlabeled Necklaces, and Irreducible Polynomials Over $\text{GF}(2)$ ”. *Journal of Algorithms*, 37(2):267–282. 2000. DOI: 10.1006/jagm.2000.1108.
 - [22] Lijun Chang, Jeffrey Xu Yu, and Lu Qin. “Fast Maximal Cliques Enumeration in Sparse Graphs”. *Algorithmica*, 66(1):173–186. 2013. DOI: 10.1007/s00453-012-9632-8.
 - [23] Y. H. Chang, Jia-Shung Wang, and Richard C. T. Lee. “Generating All Maximal Independent Sets on Trees in Lexicographic Order”. *Information Sciences*, 76(3-4):279–296. 1994. DOI: 10.1016/0020-0255(94)90013-2.
 - [24] Chandra R. Chegiredy and Horst W. Hamacher. “Algorithms for Finding K-best Perfect Matchings”. *Discrete Applied Mathematics*, 18(2):155–165. 1987. DOI: 10.1016/0166-218X(87)90017-5.
 - [25] James Cheng, Yiping Ke, Ada Wai-Chee Fu, Jeffrey Xu Yu, and Linhong Zhu. “Finding Maximal Cliques in Massive Networks”. *ACM Transactions on Database Systems*, 36(4):1–34. 2011. DOI: 10.1145/2043652.2043654.

- [26] James Cheng, Linhong Zhu, Yiping Ke, and Shumo Chu. “Fast algorithms for maximal clique enumeration with limited memory”. In *Proceedings of The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2012*, pages1240–1248. ACM, 2012. DOI: 10 . 1145 / 2339530 . 2339724.
- [27] Norishige Chiba and Takao Nishizeki. “Arboricity and Subgraph Listing Algorithms”. *SIAM Journal on Computing*, 14(1):210–223. 1985. DOI: 10 . 1137 / 0214017.
- [28] Hung Chim and Xiaotie Deng. “A New Suffix Tree Similarity Measure for Document Clustering”. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007*, pages121–130. ACM, 2007. DOI: 10 . 1145 / 1242572 . 1242590.
- [29] Michael Collins and Nigel Duffy. “Convolution Kernels for Natural Language”. In *Proceedings of Advances in Neural Information Processing Systems (NIPS 2001)*, pages625–632. NIPS, 2001.
- [30] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 2nd ed. The MIT Press, 2001.
- [31] Taishin Daigo and Kouichi Hirata. “On Generating All Maximal Acyclic Subhypergraphs With Polynomial Delay”. In *Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2009*. Lecture Notes in Computer Science. Vol. 5404, pages181–192. Springer Berlin Heidelberg, 2009. DOI: 10 . 1007 / 978 - 3 - 540 - 95891 - 8_19.
- [32] Peter Damaschke. “Parameterized Enumeration, Transversals, and Imperfect Phylogeny Reconstruction”. *Theoretical Computer Science*, 351(3):337–350. 2006. DOI: 10 . 1016 / j . tcs . 2005 . 10 . 004.

- [33] G. Danielson. “On Finding the Simple Paths and Circuits in a Graph”. *IEEE Transactions on Circuit Theory*, 15(3):294–295. 1968. DOI: 10.1109/TCT.1968.1082837.
- [34] Vânia M.F. Dias, Celina M.H. de Figueiredo, and Jayme L. Szwarcfiter. “Generating Bicliques of a Graph in Lexicographic Order”. *Theoretical Computer Science*, 337(1-3):240–248. 2005. DOI: 10.1016/j.tcs.2005.01.014.
- [35] Nan Du, Bin Wu, Liutong Xu, Bai Wang, and Xin Pei. “A Parallel Algorithm for Enumerating All Maximal Cliques in Complex Network”. In *Workshops Proceedings of the Sixth IEEE International Conference on Data Mining, ICDM 2006*), pages320–324. IEEE, 2006. DOI: 10.1109/ICDMW.2006.17.
- [36] John D. Eblen, Charles A. Phillips, Gary L. Rogers, and Michael A. Langston. “The Maximum Clique Enumeration Problem: Algorithms, Applications, and Implementations”. *BMC Bioinformatics*, 13(S-10):S5. 2012. DOI: 10.1186/1471-2105-13-S10-S5.
- [37] David Eppstein. “Finding the K Shortest Paths”. *SIAM Journal on Computing*, 28(2):652–673. 1998. DOI: 10.1137/S0097539795290477.
- [38] David Eppstein. “Finding the K Smallest Spanning Trees”. *BIT*, 32(2):237–248. 1992. DOI: 10.1007/BF01994879.
- [39] David Eppstein. “Small Maximal Independent Sets and Faster Exact Graph Coloring”. *Journal of Graph Algorithms and Applications*, 7(2):131–140. 2003. DOI: 10.7155/jgaa.00064.
- [40] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. “Sparsification—A Technique for Speeding Up Dynamic Graph Algorithms”. *Journal of the ACM*, 44(5):669–696. 1997. DOI: 10.1145/265910.265914.

- [41] David Eppstein, Maarten Löffler, and Darren Strash. “Listing All Maximal Cliques in Sparse Graphs in Near-Optimal Time”. In *Proceedings of the 21st International Symposium on Algorithms and Computation, Part I, ISAAC 2010*. Lecture Notes in Computer Science. Vol. 6506, pages403–414. Springer Berlin Heidelberg, 2010. DOI: 10.1007/978-3-642-17517-6_36.
- [42] M. C. Er. “A Note on Generating Well-Formed Parenthesis Strings Lexicographically”. *The Computer Journal*, 26(3):205–207. 1983. DOI: 10.1093/comjnl/26.3.205.
- [43] Ronald Fagin. “Degrees of Acyclicity for Hypergraphs and Relational Database Schemes”. *Journal of the ACM*, 30(3):514–550. 1983. DOI: 10.1145/2402.322390.
- [44] Michael R. Fellows, Guillaume Fertin, Danny Hermelin, and Stéphane Vialette. “Sharp Tractability Borderlines for Finding Connected Motifs in Vertex-Colored Graphs”. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming ICALP 2007*. Lecture Notes in Computer Science. Vol. 4596, pages340–351. Wroclaw, Poland, July 9-13, 2007: Springer Berlin Heidelberg, 2007. DOI: 10.1007/978-3-540-73420-8_31.
- [45] Rui Ferreira. “Efficiently Listing Combinatorial Patterns in Graphs”. PhD thesis. Università degli Studi di Pisa, Aug. 2013.
- [46] Rui A. Ferreira, Roberto Grossi, and Romeo Rizzi. “Output-Sensitive Listing of Bounded-Size Trees in Undirected Graphs”. In *Proceedings of the 19th Annual European Symposium on Algorithms, ESA 2011*. Lecture Notes in Computer Science. Vol. 6942, pages275–286. Springer Berlin Heidelberg, 2011. DOI: 10.1007/978-3-642-23719-5_24.
- [47] Rui A. Ferreira, Roberto Grossi, Romeo Rizzi, Gustavo Sacomoto, and Marie-France Sagot. “Amortized $\tilde{O}(|V|)$ -Delay Algorithm for Listing Chordless Cycles

- in Undirected Graphs”. In *Proceedings of the 22th Annual European Symposium on Algorithms, ESA 2014*. Lecture Notes in Computer Science. Vol. 8737, pages 418–429. Springer Berlin Heidelberg, 2014. DOI: 10.1007/978-3-662-44777-2_35.
- [48] Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2010. DOI: 10.1007/978-3-642-16533-7.
- [49] Greg N. Frederickson. “Data Structures for On-Line Updating of Minimum Spanning Trees, With Applications”. *SIAM Journal on Computing*, 14(4):781–798. 1985. DOI: 10.1137/0214055.
- [50] Harold Fredricksen and Irving J. Kessler. “An Algorithm for Generating Necklaces of Beads in Two Colors”. *Discrete Mathematics*, 61(2-3):181–188. 1986. DOI: 10.1016/0012-365X(86)90089-0.
- [51] Harold Fredricksen and James Maiorana. “Necklaces of Beads in K Colors and K-Ary de Bruijn Sequences”. *Discrete Mathematics*, 23(3):207–210. 1978. DOI: 10.1016/0012-365X(78)90002-X.
- [52] K. Fukuda and T. Matsui. “Finding All the Perfect Matchings in Bipartite Graphs”. *Applied Mathematics Letters*, 7(1):15–18. 1994. DOI: 10.1016/0893-9659(94)90045-0.
- [53] Komei Fukuda. *Table of Enumeration Algorithms*. 1996. URL: http://www-oldurls.inf.ethz.ch/personal/fukudak/publ/Enumeration/enumalgo95_update.pdf (visited on 12/16/2015).
- [54] Komei Fukuda and Tomomi Matsui. “Finding All Minimum-Cost Perfect Matchings in Bipartite Graphs”. *Networks*, 22(5):461–468. 1992. DOI: 10.1002/net.3230220504.

- [55] Harold N. Gabow. “Two Algorithms for Generating Weighted Spanning Trees in Order”. *SIAM Journal on Computing*, 6(1):139–150. 1977. DOI: 10.1137/0206011.
- [56] Harold N. Gabow, Zvi Galil, Thomas Spencer, and Robert Endre Tarjan. “Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs”. *Combinatorica*, 6(2):109–122. 1986. DOI: 10.1007/BF02579168.
- [57] Harold N. Gabow and Eugene W. Myers. “Finding All Spanning Trees of Directed and Undirected Graphs”. *SIAM Journal on Computing*, 7(3):280–287. 1978. DOI: 10.1137/0207024.
- [58] Alain Gély, Lhouari Nourine, and Bachir Sadi. “Enumeration Aspects of Maximal Cliques and Bicliques”. *Discrete Applied Mathematics*, 157(7):1447–1459. 2009. DOI: 10.1016/j.dam.2008.10.010.
- [59] L. Gerhards and W. Lindenberg. “Clique Detection for Nondirected Graphs: Two New Algorithms”. *Computing*, 21(4):295–322. 1979. DOI: 10.1007/BF02248731.
- [60] Dan Gusfield and Robert W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. Foundations of computing series. MIT Press, 1989.
- [61] S.L. Hakimi. “On Trees of a Graph and Their Generation”. *Journal of the Franklin Institute*, 272(5):347–359. 1961. DOI: 10.1016/0016-0032(61)90036-9.
- [62] William Hendrix, Matthew C. Schmidt, Paul Breimyer, and Nagiza F. Samatova. “On Perturbation Theory and an Algorithm for Maximal Clique Enumeration in Uncertain and Noisy Graphs”. In *Proceedings of the First ACM SIGKDD Workshop on Knowledge Discovery from Uncertain Data, U2009*, pages48–56. ACM, 2009. DOI: 10.1145/1610555.1610562.

- [63] William Hendrix, Matthew C. Schmidt, Paul Breimyer, and Nagiza F. Samatova. “Theoretical Underpinnings for Maximal Clique Enumeration on Perturbed Graphs”. *Theoretical Computer Science*, 411(26-28):2520–2536. 2010. DOI: 10.1016/j.tcs.2010.03.011.
- [64] Christopher J. Henry and Sheela Ramanna. “Maximal Clique Enumeration in Finding Near Neighbourhoods”. In *Proceedings of Transactions on Rough Sets XVI*. Lecture Notes in Computer Science. Vol. 7736, pages103–124. Springer, 2013. DOI: 10.1007/978-3-642-36505-8_7.
- [65] Teruo Hikita. “Listing and Counting Subtrees of Equal Size of a Binary Tree”. *Information Processing Letters*, 17(4):225–229. 1983. DOI: 10.1016/0020-0190(83)90046-7.
- [66] Kouichi Hirata, Megumi Kuwabara, and Masateru Harao. “On Finding Acyclic Subhypergraphs”. *Fundamentals of Computation Theory*, 491–503. 2005. DOI: 10.1007/978-3-642-19094-0_7.
- [67] Falk Hüffner, Christian Komusiewicz, Hannes Moser, and Rolf Niedermeier. “Isolation Concepts for Clique Enumeration: Comparison and Computational Experiments”. *Theoretical Computer Science*, 410(52):5384–5397. 2009. DOI: 10.1016/j.tcs.2009.05.008.
- [68] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. “An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data”. In *Proceedings of the Fourth European Conference on Principles of Data Mining and Knowledge Discovery, PKDD 2000*. Lecture Notes in Computer Science. Vol. 1910, pages13–23. Springer Berlin Heidelberg, 2000. DOI: 10.1007/3-540-45372-5_2.

- [69] Hiro Ito and Kazuo Iwama. “Enumeration of Isolated Cliques and Pseudo-Cliques”. *ACM Transactions on Algorithms*, 5(4):1–21. 2009. DOI: 10.1145/1597036.1597044.
- [70] Hiro Ito, Kazuo Iwama, and Tsuyoshi Osumi. “Linear-Time Enumeration of Isolated Cliques”. In *Proceedings of the 13th Annual European Symposium on Algorithms, ESA 2005*. Lecture Notes in Computer Science. Vol. 3669, pages119–130. Springer Berlin Heidelberg, 2005. DOI: 10.1007/11561071_13.
- [71] Jesper Jansson, Kunihiro Sadakane, and Wing-Kin Sung. “CRAM: Compressed Random Access Memory”. In *Proceedings of the 39th International Colloquium on Automata, Languages, and Programming, Part I, ICALP 2012*. Lecture Notes in Computer Science. Vol. 7391, pages510–521. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-31594-7_43.
- [72] R. Jayakumar, K. Thulasiraman, and M. Swamy. “Complexity of Computation of a Spanning Tree Enumeration Algorithm”. *IEEE Transactions on Circuits and Systems*, 31(10):853–860. 1984. DOI: 10.1109/TCS.1984.1085435.
- [73] David S. Johnson, Mihalis Yannakakis, and Christos H. Papadimitriou. “On Generating All Maximal Independent Sets”. *Information Processing Letters*, 27(3):119–123. Mar. 1988. DOI: 10.1016/0020-0190(88)90065-8.
- [74] H. C. Johnston. “Cliques of a Graph-Variations on the Bron-Kerbosch Algorithm”. *International Journal of Computer & Information Sciences*, 5(3):209–238. 1976. DOI: 10.1007/BF00991836.
- [75] T. Kamae. “A Systematic Method of Finding All Directed Circuits and Enumerating All Directed Paths”. *IEEE Transactions on Circuit Theory*, 14(2):166–171. 1967. DOI: 10.1109/TCT.1967.1082699.

- [76] Sanjiv Kapoor and H. Ramesh. “Algorithms for Generating All Spanning Trees of Undirected, Directed and Weighted graphs”. In *Proceedings of the Second Workshop on Algorithms and Data Structures, WADS 1991*. Lecture Notes in Computer Science. Vol. 519, pages461–472. Springer Berlin Heidelberg, 1991. DOI: 10.1007/BFb0028284.
- [77] Hisashi Kashima and Teruo Koyanagi. “Kernels for Semi-Structured Data”. In *Proceedings of the Nineteenth International Conference on Machine Learning, ICML 2002*, pages291–298. Morgan Kaufmann Publishers Inc., 2002.
- [78] Toshinobu Kashiwabara, Sumio Masuda, Kazuo Nakajima, and Toshio Fujisawa. “Generation of Maximum Independent Sets of a Bipartite Graph and Maximum Cliques of a Circular-Arc Graph”. *Journal of Algorithms*, 13(1):161–174. 1992. DOI: 10.1016/0196-6774(92)90012-2.
- [79] N. Katoh, T. Ibaraki, and H. Mine. “Minimum Spanning Trees”. *SIAM Journal on Computing*, 10(2):247–255. 1981. DOI: 10.1137/0210017.
- [80] Shinji Kawasoe, Hiroshi Sakamoto, Hiroki Arimura, and Setsuo Arikawa. “Efficient Substructure Discovery from Large Semi-Structured Data”. *IEICE Transactions on Information and Systems*, E87-D(12):2754–2763. 2004.
- [81] Daisuke Kimura, Tetsuji Kuboyama, Tetsuo Shibuya, and Hisashi Kashima. “A Subpath Kernel for Rooted Unordered Trees”. In *Proceedings of the 15th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, Part I, PAKDD 2011*. Lecture Notes in Computer Science. Vol. 6634, pages62–74. Springer Berlin Heidelberg, 2011. DOI: 10.1007/978-3-642-20841-6_6.
- [82] Masashi Kiyomi and Takeaki Uno. “Generating Chordal Graphs Included in Given Graphs”. *IEICE Transactions on Information and Systems*, E89-D(2):763–770. 2006. DOI: 10.1093/ietisy/e89-d.2.763.
- [83] Donald Ervin Knuth. “Dancing Links”. *eprint*, 2000. eprint: [arXiv:cs/0011047](https://arxiv.org/abs/cs/0011047).

- [84] D. Kroft. “All Paths Through a Maze”. *Proceedings of the IEEE*, 55(1):88–90. 1967. DOI: 10.1109/PROC.1967.5389.
- [85] Tetsuji Kuboyama, Kouichi Hirata, and Kiyoko F. Aoki-Kinoshita. “An Efficient Unordered Tree Kernel and Its Application to Glycan Classification”. In *Proceedings of the 12th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, PAKDD 2008*. Lecture Notes in Computer Science. Vol. 5012, pages184–195. Springer Berlin Heidelberg, 2008. DOI: 10.1007/978-3-540-68125-0_18.
- [86] Tetsuji Kuboyama, Kouichi Hirata, Hisashi Kashima, Kiyoko F. Aoki-Kinoshita, and Hiroshi Yasuda. “A Spectrum Tree Kernel”. *Information and Media Technologies*, 22(2):292–299. 2007. DOI: 10.1527/tjsai.22.140.
- [87] Taku Kudo, Eisaku Maeda, and Yuji Matsumoto. “An Application of Boosting to Graph Classification”. In *Proceedings of Advances in Neural Information Processing Systems 17, NIPS 2004*, pages729–736. NIPS, 2004.
- [88] Vincent Lacroix, Cristina G. Fernandes, and Marie-France Sagot. “Motif Search in Graphs: Application to Metabolic Networks”. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3:360–368 4. 2006. DOI: 10.1109/TCBB.2006.55.
- [89] Praveen Lakkaraju, Susan Gauch, and Mirco Speretta. “Document Similarity Based on Concept Tree Distance”. In *Proceedings of the 19th ACM Conference on Hypertext and Hypermedia, HYPERTEXT 2008*, pages127–132. ACM, 2008. DOI: 10.1145/1379092.1379118.
- [90] Joseph Y.-T. Leung. “Fast Algorithms for Generating All Maximal Independent Sets of Interval, Circular-Arc and Chordal Graphs”. *Journal of Algorithms*, 5(1):22–35. 1984. DOI: 10.1016/0196-6774(84)90037-3.

- [91] Xiao-Li Li, Soon-Heng Tan, Chuan-Sheng Foo, and See-Kiong Ng. “Interaction Graph Mining for Protein Complexes Using Local Clique Merging”. *Genome Informatics*, 16(2):260–269. 2005. DOI: 10.11234/gi1990.16.2_260.
- [92] Don R. Lick and Arthur T. White. “ K -Degenerate Graphs”. *Canadian Journal of Mathematics*, 22(5):1082–1096. 1970. DOI: 10.4153/CJM-1970-125-1.
- [93] E. Loukakis and C. Tsouros. “A Depth First Search Algorithm to Generate the Family of Maximal Independent Sets of a Graph Lexicographically”. *Computing*, 27(4):349–366. 1981. DOI: 10.1007/BF02277184.
- [94] L. Lovász. “Matroid Matching and Some Applications”. *Journal of Combinatorial Theory, Series B*, 236:208–236. 1980. DOI: 10.1016/0095-8956(80)90066-0.
- [95] Kazuhisa Makino and Takeaki Uno. “New Algorithms for Enumerating All Maximal Cliques”. In *Proceedings of the Nineth Scandinavian Workshop on Algorithm Theory, SWAT 2004*. Lecture Notes in Computer Science. Vol. 3111, pages260–272. Springer Berlin Heidelberg, 2004. DOI: 10.1007/978-3-540-27810-8_23.
- [96] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.
- [97] T. Matsui. “A Flexible Algorithm for Generating All the Spanning Trees in Undirected Graphs”. *Algorithmica*, 18(4):530–543. 1997. DOI: 10.1007/PL00009171.
- [98] Tomomi Matsui, Akihisa Tamura, and Yoshiko Ikebe. “Algorithms for Finding a K th Best Valued Assignment”. *Discrete Applied Mathematics*, 50(3):283–296. 1994. DOI: 10.1016/0166-218X(92)00175-L.
- [99] David W. Matula and Leland L. Beck. “Smallest-Last Ordering and Clustering and Graph Coloring Algorithms”. *Journal of the ACM*, 30(3):417–427. 1983. DOI: 10.1145/2402.322385.

- [100] W. Mayeda and S. Seshu. “Generation of Trees Without Duplications”. *IEEE Transactions on Circuit Theory*, 12(2):181–185. 1965. DOI: 10.1109/TCT.1965.1082432.
- [101] Ernst W. Mayr and C. Greg Plaxton. “On the Spanning Trees of Weighted Graphs”. *Combinatorica*, 12(4):433–447. 1992. DOI: 10.1007/BF01305236.
- [102] George J. Minty. “A Simple Algorithm for Listing All the Trees of a Graph”. *IEEE Transactions on Circuit Theory*, 12(1):120. 1965. DOI: 10.1109/TCT.1965.1082385.
- [103] George J. Minty. “On Maximal Independent Sets of Vertices in Claw-Free Graphs”. *Journal of Combinatorial Theory, Series B*, 28(3):284–304. 1980. DOI: 10.1016/0095-8956(80)90074-X.
- [104] Natwar Modani and Kuntal Dey. “Large Maximal Cliques Enumeration in Sparse Graphs”. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM 2008*, pages1377–1378. ACM, 2008. DOI: 10.1145/1458082.1458288.
- [105] Gordon D. Mulligan and D. G. Corneil. “Corrections to Bierstone’s Algorithm for Generating Cliques”. *Journal of the ACM*, 19(2):244–247. 1972. DOI: 10.1145/321694.321698.
- [106] Shinichi Nakano. “Efficient Generation of Plane Trees”. *Information Processing Letters*, 84(3):167–172. 2002. DOI: 10.1016/S0020-0190(02)00240-5.
- [107] Yoshio Okamoto, Takeaki Uno, and Ryuhei Uehara. “Counting the Number of Independent Sets in Chordal Graphs”. *Journal of Discrete Algorithms*, 6(2):229–242. 2008. DOI: 10.1016/j.jda.2006.07.006.
- [108] Yoshio Okamoto, Takeaki Uno, and Ryuhei Uehara. “Linear Time Counting Algorithms for Independent Sets in Chordal Graphs”. In *Proceedings of the*

- 31st International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2005*. Lecture Notes in Computer Science. Vol. 3787, pages 433–444. Springer Berlin Heidelberg, 2005. DOI: 10.1007/11604686_38.
- [109] Long Pan and Eunice E. Santos. “An Anytime-Anywhere Approach for Maximal Clique Enumeration in Social Network Analysis”. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pages 3529–3535. IEEE, 2008. DOI: 10.1109/ICSMC.2008.4811845.
 - [110] Panos M. Pardalos and Jue Xue. “The Maximum Clique Problem”. *Journal of Global Optimization*, 4(3):301–328. 1994. DOI: 10.1007/BF01098364.
 - [111] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. “Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach”. *IEEE Transactions on Knowledge and Engineering*, 16(11):1424–1440. 2004. DOI: 10.1109/TKDE.2004.77.
 - [112] J. Ponstein. “Self-Avoiding Paths and the Adjacency Matrix of a Graph”. *SIAM Journal on Applied Mathematics*, 14(3):600–609. 1966. DOI: 10.1137/0114051.
 - [113] Ronald C. Read and Robert E. Tarjan. “Bounds on Backtrack Algorithms for Listing Cycles, Paths, and Spanning Trees”. *Networks*, 5(3):237–252. 1975.
 - [114] Samuel Rota Bulò, Andrea Torsello, and Marcello Pelillo. “A Game-Theoretic Approach to Partial Clique Enumeration”. *Image and Vision Computing*, 27(7):911–922. 2009. DOI: 10.1016/j.imavis.2008.10.003.
 - [115] Doron Rotem and Julia Urrutia. “Finding Maximum Cliques In Circle Graphs”. *Networks*, 11(3):269–278. 1981. DOI: 10.1002/net.3230110305.
 - [116] Frank Ruskey. “Listing and Counting Subtrees of a Tree”. *SIAM Journal on Computing*, 10(1):141–150. 1981. DOI: 10.1137/0210011.

- [117] Frank Ruskey, Carla Savage, and Terry Min Yih Wang. “Generating Necklaces”. *Journal of Algorithms*, 13(3):414–430. 1992. DOI: 10.1016/0196-6774(92)90047-G.
- [118] Frank Ruskey and Joe Sawada. “An Efficient Algorithm for Generating Necklaces With Fixed Density”. *SIAM Journal on Computing*, 29(2):671–684. 1999. DOI: 10.1137/S0097539798344112.
- [119] Frank Ruskey and Joe Sawada. “Generating Necklaces and Strings With Forbidden Substrings”. In *Proceedings of the Sixth Annual International Conference on Computing and Combinatorics, COCOON 2000*. Lecture Notes in Computer Science. Vol. 1858, pages330–339. Springer Berlin Heidelberg, 2000. DOI: 10.1007/3-540-44968-X_33.
- [120] Kunihiko Sadakane and Hiroshi Imai. “Fast Algorithms for k-Word Proximity Search”. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E84-A(9):2311–2318. 2001.
- [121] Carla Savage. “A Survey of Combinatorial Gray Codes”. *SIAM Review*, 39(4):605–629. 1997. DOI: 10.1137/S0036144595295272.
- [122] Joe Sawada. “A Fast Algorithm to Generate Necklaces With Fixed Content”. *Theoretical Computer Science*, 301(1-3):477–489. 2003. DOI: 10.1016/S0304-3975(03)00049-5.
- [123] Matthew C. Schmidt, Nagiza F. Samatova, Kevin Thomas, and Byung-Hoon Park. “A Scalable, Parallel Algorithm for Maximal Clique Enumeration”. *Journal of Parallel and Distributed Computing*, 69(4):417–428. 2009. DOI: 10.1016/j.jpdc.2009.01.003.
- [124] Akiyoshi Shioura and Akihisa Tamura. “Efficiently Scanning all Spanning Trees of an Undirected Graph”. *Journal of the Operations Research Society of Japan*, 38(3):331–344. 1995.

- [125] Akiyoshi Shioura, Akihisa Tamura, and Takeaki Uno. “An Optimal Algorithm for Scanning All Spanning Trees of Undirected Graphs”. *SIAM Journal on Computing*, 26(3):678–692. 1997. DOI: 10.1137/S0097539794270881.
- [126] Arlei Silva, Wagner Meira, and Mohammed J. Zaki. “Structural Correlation Pattern Mining for Large Graphs”. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs, MLG 2010*, pages119–126. ACM, 2010. DOI: 10.1145/1830252.1830268.
- [127] Kenneth Sörensen and Gerrit K. Janssens. “An Algorithm to Generate All Spanning Trees of a Graph in Order of Increasing Cost”. *Pesquisa Operacional*, 25(2):219–229. 2005. DOI: 10.1590/S0101-74382005000200004.
- [128] Volker Stix. “Finding All Maximal Cliques in Dynamic Graphs”. *Computational Optimization and Applications*, 27(2):173–186. 2004. DOI: 10.1023/B:COAP.0000008651.28952.b6.
- [129] Hisao Tamaki. “Space-Efficient Enumeration of Minimal Transversals of a Hypergraph”. Japanese. In *Proceedings of IPSJ SIG Notes*. 103, pages29–36. Information Processing Society of Japan, 2000.
- [130] Robert Endre Tarjan. “Enumeration of the Elementary Circuits of a Directed Graph”. *SIAM Journal on Computing*, 2(3):211–216. 1973. DOI: 10.1137/0202017.
- [131] Robert Endre Tarjan and Mihalis Yannakakis. “Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs”. *SIAM Journal on Computing*, 13(3): 1984. DOI: 10.1137/0213035.
- [132] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. “The Worst-Case Time Complexity for Generating All Maximal Cliques and Computational Experi-

- ments”. *Theoretical Computer Science*, 363(1):28–42. 2006. DOI: 10.1016/j.tcs.2006.06.015.
- [133] Koji Tsuda and Taku Kudo. “Clustering Graphs by Weighted Substructure Mining”. In *Proceedings of the 23th International Conference on Machine Learning, ICML 2006*, pages953–960. ACM, 2006. DOI: 10.1145/1143844.1143964.
- [134] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. “A New Algorithm for Generating All the Maximal Independent Sets”. *SIAM Journal on Computing*, 6(3):505–517. 1977. DOI: 10.1137/0206036.
- [135] Esko Ukkonen. “On Approximate String Matching”. In *Proceedings of the 1983 International Fundamentals of Computation Theory Conference*. Lecture Notes in Computer Science. Vol. 158, pages487–495. Springer Berlin Heidelberg, 1983. DOI: 10.1007/3-540-12689-9_129.
- [136] Takeaki Uno. “A Fast Algorithm for Enumeration of Maximal Matchings in General Graphs”. *Journal of National Institute of Informatics*, 3:89–97. 2001.
- [137] Takeaki Uno. “Algorithms for Enumerating All Perfect, Maximum and Maximal Matchings in Bipartite Graphs”. In *Proceedings of the Eighth International Symposium on Algorithms and Computation, ISAAC 1997*. Lecture Notes in Computer Science. Vol. 1350, pages92–101. Springer Berlin Heidelberg, 1997. DOI: 10.1007/3-540-63890-3_11.
- [138] Takeaki Uno. “An Algorithm for Enumerating All Directed Spanning Trees in a Directed Graph”. In *Proceedings of the Seventh International Symposium on Algorithms and Computation, ISAAC 1996*. Lecture Notes in Computer Science. Vol. 1178, pages166–173. Springer Berlin Heidelberg, 1996. DOI: 10.1007/BFb0009492.
- [139] Takeaki Uno. “An Efficient Algorithm for Solving Pseudo Clique Enumeration Problem”. *Algorithmica*, 56(1):3–16. 2010. DOI: 10.1007/s00453-008-9238-3.

- [140] Takeaki Uno. “An Output Linear Time Algorithm for Enumerating Chordless Cycles”. Japanese. *the 92nd SIGAL of Information Processing Society Japan*, 1–7. 2003.
- [141] Takeaki Uno. “Constant Time Enumeration by Amortization”. In *Proceedings of the 14th International Symposium on Algorithms and Data Structures, WADS 2015*. Lecture Notes in Computer Science. Vol. 9214, pages 593–605. Springer International Publishing, 2015. DOI: 10.1007/978-3-319-21840-3_49.
- [142] Takeaki Uno. “New Approach for Speeding Up Enumeration Algorithms”. In *Proceedings of the Ninth International Symposium on Algorithms and Computation, ISAAC 1998*. Lecture Notes in Computer Science. Vol. 1533, pages 287–296. Springer Berlin Heidelberg, 1998. DOI: 10.1007/3-540-49381-6_31.
- [143] Takeaki Uno. *Two General Methods to Reduce Delay and Change of Enumeration Algorithms*. Tech. rep. NII-2003-004E. National Institute of Informatics, 2003.
- [144] Takeaki Uno and Hiroki Arimura. “Ambiguous Frequent Itemset Mining and Polynomial Delay Enumeration”. In *Proceedings of the 12th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, PAKDD 2008*. Lecture Notes in Computer Science. Vol. 5012, pages 357–368. Springer Berlin Heidelberg, 2008. DOI: 10.1007/978-3-540-68125-0_32.
- [145] Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. “An Efficient Algorithm for Enumerating Closed Patterns in Transaction Databases”. In *Proceedings of the Seventh International Conference on Discovery Science, DS 2004*. Lecture Notes in Computer Science. Vol. 3245, pages 16–31. Springer Berlin Heidelberg, 2004. DOI: 10.1007/978-3-540-30214-8_2.
- [146] Takeaki Uno and Hiroko Satoh. “An Efficient Algorithm for Enumerating Chordless Cycles and Chordless Paths”. In *Proceedings of the 17th International*

- Conference on Discovery Science, DS 2014*. Lecture Notes in Computer Science. Vol. 8777, pages313–324. Springer International Publishing, 2014. DOI: 10.1007/978-3-319-11812-3_27.
- [147] Timothy R. Walsh. “Generation of Well-Formed Parenthesis Strings in Constant Worst-Case Time”. *Journal of Algorithms*, 29(1):165–173. 1998. DOI: 10.1006/jagm.1998.0960.
- [148] Terry Min Yih Wang and Carla D. Savage. “A Gray Code for Necklaces of Fixed Density”. *SIAM Journal on Discrete Mathematics*, 9(4):654–673. 1996. DOI: 10.1137/S089548019528143X.
- [149] Kunihiro Wasa. *Enumeration of Enumeration Algorithms and Its Complexity*. 2015. URL: http://www-ikn.ist.hokudai.ac.jp/~wasa/enumeration_complexity.html (visited on 12/16/2015).
- [150] Kunihiro Wasa, Hiroki Arimura, Kouichi Hirata, and Takeaki Uno. “Faster Algorithms for Tree Similarity Based on Compressed Enumeration of Bounded-Sized Ordered Subtrees”. In *Proceedings of the Sixth International Conference on Similarity Search and Applications, SISAP 2013*. Lecture Notes in Computer Science. Vol. 8199, pages73–84. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-41062-8_8.
- [151] Kunihiro Wasa, Hiroki Arimura, and Takeaki Uno. “Efficient Enumeration of Induced Subtrees in a K-Degenerate Graph”. In *Proceedings of the 25th International Symposium on Algorithms and Computation, ISAAC 2014*. Lecture Notes in Computer Science. Vol. 6506, pages94–102. Springer International Publishing, 2014. DOI: 10.1007/978-3-319-13075-0_8.
- [152] Kunihiro Wasa, Yusaku Kaneta, Takeaki Uno, and Hiroki Arimura. “Constant Time Enumeration of Bounded-Size Subtrees in Trees and Its Application”. In *Proceedings of the 18th Annual International Conference on Computing and*

- Combinatorics, COCOON 2012*. Lecture Notes in Computer Science. Vol. 7434, pages 347–359. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-32241-9_30.
- [153] Kunihiro Wasa, Takeaki Uno, Kouichi Hirata, and Hiroki Arimura. “Polynomial Delay and Space Discovery of Connected and Acyclic Sub-Hypergraphs in a Hypergraph”. In *Proceedings of the 16th International Conference on Discovery Science, DS 2013*. Lecture Notes in Computer Science. Vol. 8140, pages 308–323. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-40897-7_21.
- [154] Marcel Wild. “Output-Polynomial Enumeration of All Fixed-Cardinality Ideals of a Poset, Respectively All Fixed-Cardinality Subtrees of a Tree”. *Order*, 31(1):121–135. 2013. DOI: 10.1007/s11083-013-9292-6.
- [155] Dong Xin, Jiawei Han, Xifeng Yan, and Hong Cheng. “Mining Compressed Frequent-Pattern Sets”. In *Proceedings of the 31st International Conference on Very Large Data Bases, VLDB 2005*, pages 709–720. ACM, 2005.
- [156] Takeo Yamada, Seiji Kataoka, and Kohtaro Watanabe. “Listing All the Minimum Spanning Trees in an Undirected Graph”. *International Journal of Computer Mathematics*, 87(14):3175–3185. 2010. DOI: 10.1080/00207160903329699.
- [157] Xifeng Yan and Jiawei Han. “gSpan: Graph-Based Substructure Pattern Mining”. In *Proceedings of the 2002 IEEE International Conference on Data Mining, ICDM 2002*, pages 721–724. IEEE, 2002. DOI: 10.1109/ICDM.2002.1184038.
- [158] S. Yau. “Generation of all Hamiltonian Circuits, Paths, and Centers of a Graph, and Related Problems”. *IEEE Transactions on Circuit Theory*, 14(1):79–81. 1967. DOI: 10.1109/TCT.1967.1082662.
- [159] Yasuko Yoshida. *List of Enumeration Algorithms*. 1993. URL: <http://www-oldurls.inf.ethz.ch/personal/fukudak/publ/Enumeration/enumalgo93.pdf> (visited on 12/16/2015).

- [160] M.J. Zaki and C.-J. Hsiao. “Efficient Algorithms for Mining Closed Itemsets and Their Lattice Structure”. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):462–478. 2005. DOI: 10.1109/TKDE.2005.60.
- [161] Mohammed Javeed Zaki. “Efficiently Mining Frequent Trees in a Forest”. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2002*, pages71–80. ACM, 2002. DOI: 10.1145/775047.775058.
- [162] Mohammed Javeed Zaki. “Scalable Algorithms for Association Mining”. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390. 2000. DOI: 10.1109/69.846291.